GÁBOR BEREND

DATA MINING

Lectored by: Gábor Németh, Dr.

This teaching material has been made at the University of Szeged, and supported by the European Union. Project identity number: EFOP-3.4.3-16-2016-00014

Table of Contents

1 *About this book* 15

1.1 *How to Use this Book* 15

2 *Programming background* 17

2.1 *Octave* 17

2.2 *Plotting* 26

2.3 *A brief comparison of Octave and numpy* 29

2.4 *Summary of the chapter* 30

3 *Basic concepts* 31

3.1 *Goal of data mining* 32

3.2 *Representing data* 38

3.3 *Information theory and its application in data mining* 46

3.4 *Eigenvectors and eigenvalues* 50

3.5 *Summary of the chapter* 54

4 *Distances and similarities* 55

4.1 *Minkowski distance* 57

4.2 *Mahalanobis distance* 60

4.3 *Cosine distance* 62

4.4 *Jaccard similarity* 64

4.5 *Edit distance* 66

4.6 *Distances for distributions* 68
4.7 *Euclidean versus non-Euclidean distances* 73
4.8 *Summary of the chapter* 75

5 *Finding Similar Objects Efficiently* 76
5.1 *Locality Sensitive Hashing* 76
5.2 *Approaches disallowing false negatives* 90
5.3 *Summary of the chapter* 99

6 *Dimensionality reduction* 100
6.1 *The curse of dimensionality* 101
6.2 *Principal Component Analysis* 105
6.3 *Singular Value Decomposition* 118
6.4 *Linear Discriminant Analysis* 131
6.5 *Further reading* 135
6.6 *Summary of the chapter* 137

7 *Mining frequent item sets* 138
7.1 *Important concepts and notation for frequent pattern mining* 139
7.2 *Apriori algorithm* 146
7.3 *Park–Chen–Yu algorithm* 155
7.4 *FP–Growth* 159
7.5 *Summary of the chapter* 166

8 *Data mining from networks* 168
8.1 *Graphs as complex networks* 168
8.2 *Markov processes* 171
8.3 *PageRank algorithm* 177
8.4 *Hubs and Authorities* 186
8.5 *Further reading* 189
8.6 *Summary of the chapter* 190

9 *Clustering* 191
9.1 *What is clustering?* 191
9.2 *Agglomerative clustering* 194
9.3 *Partitioning clustering* 200
9.4 *Further reading* 208
9.5 *Summary of the chapter* 209

10 *Conclusion* 210

A *Appendix — Course information* 211

Bibliography 215

List of Figures

2.1 A screenshot from the graphical interface of Octave. 18
2.2 Code snippet: *Hello world!* in Octave 19
2.3 Code snippet: An example for using a for loop and conditional execution with an if construction to print even numbers between 1 and 10. 19
2.4 Code snippet: The general syntax for defining a function 20
2.5 Code snippet: A simple function receiving two matrices and returning both their sum and difference. 20
2.6 Code snippet: An example of a simple anonymous function which returns the squared sum of its arguments (assumed to be scalars instead of vectors). 21
2.7 Code snippet: An illustration of indexing matrices in Octave. 22
2.8 Code snippet: Centering the dataset with `for` loop 23
2.9 Code snippet: Centering the dataset relying on broadcasting 23
2.10 Code snippet: Calculating the columnwise sum of a matrix with a `for` loop. 24
2.11 Code snippet: Calculating the columnwise sum of a matrix in a vectorized fashion *without* a `for` loop. 25
2.12 Code snippet: Calculating the columnwise sum of a matrix in a semi-vectorized fashion with single a `for` loop. 25
2.13 Code snippet: Transforming vectors in a matrix to unit-norm by relying on a `for` loop. 26
2.14 Code snippet: Transforming vectors in a matrix to unit-norm in a vectorized fashion *without* a `for` loop. 26
2.15 Code snippet: Code for drawing a simple function $f(x) = 2x^3$ over the range $[-3, 4]$. 26
2.16 The resulting output of the code snippet in Figure 2.15 for plotting the function $f(x) = 2x^3$ over the interval $[-3, 4]$. 27
2.17 Code snippet: Code for drawing a scatter plot of the heights of the people from the sample to be found in Table 2.1. 28
2.18 Scatter plot of the heights and weights of the people from the sample to be found in Table 2.1. 28

2.19 Code snippet: Code for drawing a histogram of the heights of the people from the sample to be found in Table 2.1. 28
2.20 Histogram of the heights of the people from the sample to be found in Table 2.1. 29
2.21 Code snippet: Comparing matrix-style and elementwise multiplication of matrices. 30

3.1 The process of knowledge discovery 33
3.2 Example of a spurious correlation. Original source of the plot: `http://www.tylervigen.com/spurious-correlations` 33
3.3 Code snippet: Various preprocessing steps of a bivariate dataset 41
3.4 Various transformations of the originally not origin centered and correlated dataset (with the thick orange cross as the origin). 42
3.5 Math Review: Scatter and covariance matrix 44
3.6 Math Review: Cholesky decomposition 45
3.7 Math Review: Shannon entropy 47
3.8 The amount of surprise for some event as a function of the probability of the event. 48
3.9 Sample feature distribution with observations belonging into two possible classes (indicated by blue and red). 50
3.10 Code snippet: Eigencalculation using Octave 53
3.11 Code snippet: An alternative way of determining the eigenpairs of matrix M without relying on the built-in Octave function `eig`. 54

4.1 Illustration for the symmetric set difference fulfilling the triangle inequality. 57
4.2 Unit circles for various p norms. 58
4.3 Example points for illustrating the Minkowski distance for different values of p. 59
4.4 Motivation for the Mahalanobis distance. 60
4.5 Illustration of the Mahalanobis distance. 61
4.6 Illustration of the cosine distance 62
4.7 Math Review: Law of cosines 63
4.8 An example pair of sets X and Y. 65
4.9 The visualization of the Bhattacharyya coefficient for a pair of Bernoulli distributions P and Q. 69
4.10 Illustration for the Bhattacharyya distance not obeying the property of triangle inequality. 70
4.11 Illustration for the Hellinger distance for the distributions from Example 4.7. 71
4.12 Visualization of the distributions regarding the spare time activities in Example 4.8. R, C and H along the x-axis refers to the activities reading, going to the cinema and hiking, respectively. 73

4.13 Code snippet: Sample code to calculate the difference distance/divergence values for the distributions from Example 4.8. 74

5.1 Code snippet: Creating the characteristic matrix of the sets 77
5.2 Code snippet: Creating a random permutation of the characteristic matrix 78
5.3 Code snippet: Determine the minhash function for the characteristic matrix 78
5.4 Code snippet: Determine the minhash function for the characteristic matrix 81
5.5 Illustration of Locality Sensitive Hashing. The two bands colored red indicates that the minhash for those bands are element-wise identical. 83
5.6 Illustration of the effects of choosing different band and row number during Locality Sensitive Hashing for the Jaccard distance. Each subfigure has the number of rows per band fixed to one of $\{1, 3, 5, 7\}$ and the values for b range in the interval $[1, 10]$. 84
5.7 The ideal behavior of our hash function with respect its probability for assigning objects to the same bucket, if our threshold of critical similarity is set to 0.6. 85
5.8 The probability of assigning a pair of objects to the same basket with different number of bands and rows per bands. 85
5.9 The probability curves when performing AND-construction ($r = 3$) followed by OR-constructions ($b = 4$). The curves differ in the order the AND,-and OR-constructions follow each other. 87
5.10 Code snippet: Illustration of combining AND/OR-amplifications in different order. 87
5.11 Math Review: Dot product 88
5.12 Illustration of the geometric probability defined in Eq. (5.7). Randomly drawn hyperplanes s_1 and s_2 are denoted by dashed lines. The range of the separating angle and one such hyperplane (s_1) is drawn in red, whereas the non-separating range of angle and one such hyperplane (s_2) are in blue. 89
5.13 Approximation of the angle enclosed by the two vectors as a function of the random projections employed. The value of the actual angle is represented by the horizontal line. 90
5.14 Math Review: Euler's constant 96
5.15 Illustration of the Euler coefficient as a limit of the sequence $(1 + \frac{1}{n})^n$. 97
5.16 False positive rates of a bloom filter with different load factors as a function of the hash functions employed per inserted instances. The different load factors corresponds to different average number of objects per a bucket (G/m). 98

6.1 Gamma function over the interval [0.01,6] with integer values denoted with orange crosses. 101
6.2 Math Review: Gamma function 102
6.3 The volume of a unit hypersphere as a function of the dimensionality d. 102
6.4 The relative volume of the $1-\epsilon$ sphere to the unit sphere. 103
6.5 The position of a (red) hypersphere which touches the (blue) hyperspheres with radius 1. 103
6.6 The radius of the hypersphere in d dimensions which touches the 2^d non-intersecting unit hyperspheres located in a hypercube with sides of length of 4. 104
6.7 The distribution of the pairwise distances between 1000 pairs of points in different dimensional spaces. 105
6.8 Math Review: Constrained optimization 108
6.9 An illustration of the solution of the constrained minimization problem for $f(x,y) = 12x + 9y$ with the constraint that the solution has to lie on the unit circle. 109
6.10 Math Review: Eigenvalues of scatter and covariance matrices 111
6.11 Code snippet: Calculating the 1–dimensional PCA representation of the dataset stored in matrix M. 112
6.12 A 25 element sample from the 5,000 face images in the dataset. 113
6.13 Differently ranked eigenfaces reflecting the decreasing quality of eigenvectors as their corresponding eigenvalues decrease. 114
6.14 Visualizing the amount of distortion when relying on different amount of top-ranked eigenvectors. 115
6.15 Math Review: Eigendecomposition 119
6.16 Math Review: Frobenius norm 120
6.17 Code snippet: Performing eigendecomposition of a diagonalizable matrix in Octave 121
6.18 Visual representation of the user rating dataset from Table 6.2 in 3D space. 123
6.19 The reconstructed movie rating dataset based on different amount of singular vectors. 125
6.20 Code snippet: An illustration that the squared sum of singular values equals the squared Frobenius norm of a matrix. 126
6.21 The latent concept space representations of users and movies. 128
6.22 Example dataset for CUR factorization. The last row/column includes the probabilities for sampling the particular vector. 130
6.23 An illustration of a tensor. 131
6.24 An illustration of the effect of optimizing the joint fractional objective of LDA (b) and its nominator (c) and denominator (d) separately. 134
6.25 Code snippet: Code snippet demonstrating the procedure of LDA. 135
6.26 Illustration of the effectiveness of locally linear embedding when applied on a non-linear dataset originating from an S-shaped manifold. 136

7.1 An example Hasse diagram for items *a*, *b* and *c*. Item sets marked by red are frequent. 141
7.2 Illustration of the distinct potential association rules as a function of the different items/features in our dataset (*d*). 144
7.3 The relation of frequent item sets to closed and maximal frequent item sets. 146
7.4 Code snippet: Performing counting of single items for the first pass of the Apriori algorithm. 154
7.5 Code snippet: Performing counting of single items for the first pass of the Park-Chen-Yu algorithm. 158
7.6 The FP-tree over processing the first five baskets from the example transactional dataset from Table 7.7. The red dashed links illustrate the pointers connecting the same items for efficient aggregation. 163
7.7 FP-trees obtained after processing the entire sample transactional database from Table 7.7 when applying different item ordering. Notice that the pointers between the same items are not included for better visibility. 164
7.8 FP-trees conditionated on item set `{E}` with different ordering of the items applied. 166

8.1 A sample directed graph with four vertices (a) and its potential representations as an adjacency matrix (b) and an edge list (c). 170
8.2 Code snippet: Creating a dense and a sparse representation for the example digraph from Figure 8.1. 170
8.3 Math Review: Eigenvalues of stochastic matrices 172
8.4 An illustration of the Markov chain given by the transition probabilities from 8.2. The radii of the nodes – corresponding to the different states of our Markov chain – if proportional to their stationary distribution. 176
8.5 Code snippet: The uniqueness of the stationary distribution and its global convergence property is illustrated by the fact that all 5 random initializations for $\mathbf{p}$ converged to a very similar distribution. 176
8.6 An example network and its corresponding state transition matrix. 179
8.7 An example network which is not irreducible with its state transition matrix. The problematic part of the state transition matrix is in red. 181
8.8 An example network which is not aperiodic with its state transition matrix. The problematic part of the state transition matrix is in red. 182
8.9 An example (non-ergodic) network in which we perform restarts favoring state 1 and 2 (indicated by red background). 185
8.10 Example network to perform HITS algorithm on. 189

9.1 A schematic illustration of clustering. 192

9.2 Illustration of the hierarchical cluster structure found for the data points from Table 9.1. 197
9.3 An illustration of the k-means algorithm on a sample dataset with $k = 3$ with the trajectory of the centroids being marked at the end of different iterations. 203
9.4 Using k-means clustering over the example Braille dataset from Figure 9.1. Ideal clusters are indicated by grey rectangles. 204
9.5 Math Review: Variance as a difference of expectations 207

List of Tables

2.1 Example dataset containing height and weight measurements of people. 27

3.1 Example dataset illustrating Simpson's paradox 34
3.2 Typical naming conventions for the rows and columns of datasets. 38
3.3 Overview of the different measurement scales. 38
3.4 Illustration of the possible treatment of a nominal attribute. The newly introduced capitalized columns correspond to binary features indicating whether the given instance belongs to the given nationality, e.g., whenever the BRA variable is 1, the given object is Brazilian. 40
3.5 An example contingency table for two variables. 46
3.6 Sample dataset for illustrating feature discretization using mutual information. 50

4.1 Various distances between the distribution regarding Jack's leisure activities and his colleagues. The smaller values for each notions of distances is highlighted in bold. 73

5.1 Step-by-step derivation of the calculation of minhash values for a fixed permutation. 80

6.1 Example user-item dataset with a lower than observed effective dimensionality. 105
6.2 Example rating matrix dataset. 122
6.3 The cosine similarities between the 2-dimensional latent representations of user Anna and the other users. 127
6.4 The predicted rating given by Anna to the individual movies based on the top-2 singular vectors of the rating matrix. 127
6.5 The values of the different components of the LDA objective (along the columns) assuming that we are optimizing towards certain parts of the objective (indicated at the beginning of the rows). Best values along each column are marked bold. 135

7.1 Example transactional dataset. 139

7.2 Compressing frequent item sets — Example (t = 3) 147
7.3 Sample transactional dataset (a) and its explicit and its corresponding incidence matrix representation (b). The incidence matrix contains 1 for such combinations of baskets and items for which the item is included in the particular basket. 153
7.4 The upper triangular of the table lists virtual item pair identifiers obtained by the hash function $h(x, y) = (5(x + y) - 3 \mod 7) + 1$ for the sample transactional dataset from Table 7.3. The lower triangular values are the actual supports of all the item pairs, with the values in the diagonal (in parenthesis) include supports for item singletons. 157
7.5 The values stored in the counters created during the first pass of the Park-Chen-Yu algorithm over the sample transactional dataset from Table 7.3. 157
7.6 Support values for C_2 calculated by the end of the second pass of PCY. 159
7.7 Example market basket database. 162
7.8 The transactional dataset indicating irrelevant transactions with respect item set {E}. Irrelevant baskets are striked. 165

8.1 The convergence of the PageRank values for the example network from Figure 8.6. Each row corresponds to a vertex from the network. 180
8.2 Illustration of the power iteration for the periodic network from Figure 8.8. The cycle (consisting of a single vertex) accumulates all the importance within the network. 183
8.3 The probability of restarting a random walk after varying number of steps and damping factor β. 184
8.4 Illustration of the power iteration for the periodic network from Figure 8.8 when a damping factor $\beta = 0.8$ is applied. 184
8.5 Illustration of the convergence of the HITS algorithm for the example network included in Figure 8.10. Each row corresponds to a vertex from the network. 189

9.1 Example 2-dimensional clustering dataset. 196
9.2 Pairwise cluster distances during the execution of hierarchical clustering. The upper and lower triangular of the matrix includes the between cluster distances obtained when using complete linkage and single linkage, respectively. Boxed distances indicate the pair of clusters that get merged in a particular step of hierarchical clustering. 197
9.3 Illustration of the calculation of the medoid of a cluster. 198
9.4 An example where matrix of within-cluster distances for which different ways of aggregation yields different medoids. 199
9.5 Calculations involved in the determination of biased sample variance in two different ways for the observations $[3, 5, 3, -1]$. 207

9.6 Example 2-dimensional clustering dataset for illustrating the working mechanism of the Bradley-Fayyad-Reina algorithm. 208
9.7 Illustration of the BFR representations of clusters $C_1 = \{A, B, C\}$ and $C_2 = \{D, E\}$ for the data points from Table 9.6. 209

1 | About this book

This book was primarily written as a teaching material for computer science students at the University of Szeged taking the course and/or interested in the field of Data Mining. Readers are expected to develop an understanding on how the most popular data mining algorithms operate and under what circumstances they can be applied to process large-scale ubiquitous data. The book also provides mathematical insights, so that readers are expected to develop an ability to access the algorithms from an analytical perspective as well.

1.1 How to Use this Book

The field of data mining is so diverse that trying to cover all its aspects would be infeasible by all means. Instead of aiming at exhaustiveness, the goal of the book is to provide a self-contained selection of important concepts and algorithms related some of the crucial problems of data mining. The book is intended to illustrate the concepts both from mathematical and programming perspective as well. It covers and distills selected topics from two highly recommended textbooks:

- Jure Leskovec, Anand Rajaraman, Jeff Ullman: Mining of Massive Datasets[1],
- Pang-Ning Tan, Michael Steinbach, Vipin Kumar: Introduction to Data Mining[2]

[1] Leskovec et al. 2014

[2] Tan et al. 2005, Leskovec et al. 2014

The book assumes only a minimal amount of prerequisites in the form of basic linear algebra and calculus. If you are comfortable with the concept of vectors, matrices and the basics of derivation, you are ready to start reading the book.

Mathematical concepts that go beyond the minimally assumed knowledge are going to be introduced near the place where they are referenced. If you feel yourself comfortable with the concepts discussed in the *Math review* sections of the book, it is a safe choice to skip them and occasionally revisit them if you happen to struggle

with recalling the concepts they include. In order to increase the visibility of important technical concepts **highlighting** is employed. Highlighted concepts are also those that are included in the index at the end of the document.

Chapter 2 provides a gentle introduction to GNU **Octave**, which is a **MATLAB**-like interpreted programming language best suited for numerical computations. Throughout the book, you will find Octave code snippets in order to provide a better understanding of the concepts and algorithms being discussed. The same advise holds for Chapter 2 as well for the *Math review* sections, i.e. if you have reasonable familiarity with using Octave (or MATLAB perhaps), you can decide to skip that section without any unpleasant feelings.

Chapter 3 defines the scope of data mining and also a few important concepts related to it. Upcoming chapters of the book deal with one specific topic at a time, such as measuring and approximating similarity efficiently, dimensionality reduction, frequent pattern mining, among others.

2 | Programming background

Learning Objectives:
- Getting to know Octave
- Learning about broadcasting
- Understanding vectorization

This chapter gives an overview to the Octave programming language, which provides a convenient environment for illustrating a wide range of data mining concepts and techniques. The readers of this chapter will

- understand the programming paradigm used by Octave,
- comprehend and apply the frequently used concept of slicing, broadcasting and vectorization during writing code in Octave,
- be able to produce simple visualization for datasets,
- become aware about the possible alternatives to using Octave.

2.1 *Octave*

Scientific problems are typically solved by relying on **numerical analysis** techniques involving calculations with matrices. Implementing such routines would be extremely cumbersome and requires a fair amount of expertise in numerical methods in order to come up with fast, scalable and efficient solutions which also provide numerically stable results.

Luckily, there exist a number of languages which excel in these areas, for example **MATLAB**, GNU **Octave**, **Scilab** and **Maple**, just to name a few of them. Out of these alternatives, MATLAB probably has the most functionalities but it comes at a price since it is a proprietary software. Octave, on the other hand offers nearly as much functionality as MATLAB does with the additional benefit that it is maintained within the open source GNU ecosystem. The open source nature of GNU Octave [1] is definitely a huge benefit. GNU Octave enjoys the widest support among the open source alternatives of MATLAB. The syntax of the two languages are nearly identical, with a few differences[2].

[1] John W. Eaton, David Bateman, Søren Hauberg, and Rik Wehbring. *GNU Octave version 4.2.0 manual: a high-level interactive language for numerical computations*, 2016. URL http://www.gnu.org/software/octave/doc/interpreter

[2] https://en.wikibooks.org/wiki/MATLAB_Programming/Differences_between_Octave_and_MATLAB

2.1.1 *Getting Octave*

Octave can be downloaded from the following URL: `https://www.gnu.org/software/octave/download.html`. After installation, a simple terminal view and a GUI gets installed as well. As depicted in Figure 2.1, a central component in the graphical interface is the *Command Window*. The *Command Window* can be used invoke commands and calculate them on-the-fly. The graphical interface also incorporates additional components such as the *File browser*, *Workspace* and *Command History* panels for ease of use.

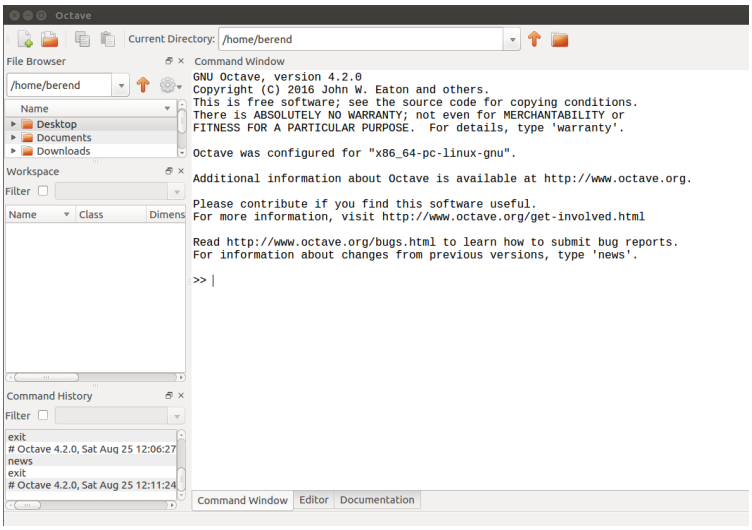


Figure 2.1: A screenshot from the graphical interface of Octave.

Even though the standalone version of Octave offers a wide range of functionalities in terms of working with matrices and plotting, it can still happen that the core functionality of Octave is not enough for certain applications. In that case the Octave-Forge `https://octave.sourceforge.io/` project library is the right place to look for any additional extensions, which might fulfill our special needs. In case we find some of the extra packages interesting all we need to do is invoking the command

```
pkg install -forge package_name
```

in an Octave terminal, where `package_name` is the name of the additional package that we want to obtain.

2.1.2 *The basic syntax of Octave*

First things first, start with the program everyone writes first when familiarizing with a new programming language, *Hello world!*. All what this simple code snippet does is that it prints the text *Hello world!* to the console. The command can simply be written in the Octave terminal and we will immediately see its effect thanks to the

interpreted nature of Octave.

CODE SNIPPET

```
printf('Hello world!\n')
>>Hello world!
```

Figure 2.2: *Hello world!* in Octave

According to the Octave philosophy, whenever something can be expressed as a matrix, think of it as such and express it as a matrix. For instance if we want to iterate over a range of n integers, we can do it so by iterating over the elements of a vector (basically a matrix of size $1 \times n$) as it is illustrated in Figure 2.3. Figure 2.3 further reveals how to use conditional expressions.

CODE SNIPPET

```
for k=1:10
    if mod(k, 2) == 0
        printf("%d ", k)
    endif
endfor
printf('\n')
>>2 4 6 8 10
```

Figure 2.3: An example for using a for loop and conditional execution with an if construction to print even numbers between 1 and 10.

Note the `1:10` notation in Figure 2.3 which creates a vector with elements ranging from 1 to 10 (both inclusive). This is the manifestation of the general structure looking `start:delta:end`, which generates a vector constituting of the members of an arithmetic series with its first element being equal to `start`, the difference between two consecutive elements being `delta` and the last element not exceeding `end`. In the absence of an explicit specification of the `delta` value it is assumed to be 1.

? Can you write an Octave code which has equivalent functionality to the one seen in Figure 2.3, but which does not use conditional if statement?

2.1.3 *Writing functions*

Just like in most programming languages, functions are an essential component of Octave. Nonetheless Octave delivers a wide range of already defined mostly mathematical functions, such as `cos()`, `sin()`, `exp()`, `roots()`, etc., it is important to know how to write our custom functions.

The basic syntax of writing a function is summarized in Figure 2.4. Every function that we define on our own has to start with the keyword `function` which can be followed by a list of variables we would like our function to return. Note that this construction offers us the

flexibility of listing more variables to return at a time. Once we defined which variables are expected to be returned, we have to give our function a unique name that we would like to reference it, and list its arguments as it can be seen in Figure 2.4. After that, we are free to do whatever calculations we would like our function to perform. All we have to make sure that by the time calculations expressed in the body of the function get executed, the variables that we have identified as the ones that would be returned get assigned the correct values according to the function. We shall indicate the end of a function by using the `endfunction` keyword.

CODE SNIPPET

```
function [return-variable(s)] = function_name (arg-list)
    body
endfunction
```

Figure 2.4: The general syntax for defining a function

Figure 2.5 provides an example realization of the general schema for defining a function provided in Figure 2.4. This simple function returns the sum and the difference of its two arguments.

CODE SNIPPET

```
function [result_sum, result_diff] = add_n_subtract(x, y)
  if all(size(x)==size(y))
    result_sum=x+y;
    result_diff=x-y;
  else
    fprintf('Arguments of incompatible sizes.\n')
    result_sum=result_diff=inf;
  endif
endfunction

[vs, vd] = add_n_subtract([4, 1], [2, -3])
>> vs = 6 -2
   vd = 2  4
[vs, vd] = add_n_subtract([4, 1], [2, -3, 5])
>> Arguments of incompatible sizes.
   vs = Inf
   vd = Inf
```

Figure 2.5: A simple function receiving two matrices and returning both their sum and difference.

Since it only makes sense to perform arithmetic operations on operands with compatible sizes (i.e. both of the function arguments has to be of the same size), we check this property of the arguments

in the body of the function in Figure 2.5. If the sizes of the arguments match each other exactly, then we perform the desired operations, otherwise we inform the user about the incompatibility of the sizes of the arguments and return with such values (infinity) which tell us that the function could not be evaluated properly. In Section 2.1.5 we will see that Octave is not that strict about arithmetic operations due to its broadcasting mechanism which tries to perform an operation even when the sizes of its arguments are not directly compatible with each other. Finally, the last two commands in Figure 2.5 also illustrate how to retrieve multiple values from such a function which returns more than just one variable at a time.

Besides the previously seen ways of defining functions, Octave offers another convenient way for it, through the use of **anonymous functions**. Anonymous functions are similar to what are called as **lambda expressions** in other programming languages. Anonymous functions according to the documentation of Octave "are useful for creating simple unnamed functions from expressions or for wrapping calls to other functions". A sample anonymous function can be found in Figure 2.6.

CODE SNIPPET

```
squared_sum = @(x,y) x^2 + y^2;
squared_sum(-2,3)
>>13
```

Figure 2.6: An example of a simple anonymous function which returns the squared sum of its arguments (assumed to be scalars instead of vectors).

2.1.4 *Arithmetics and indexing*

As mentioned earlier, due to the Octave philosophy variables are primarily treated as matrices. As such, the $*$ operator refers to matrix multiplication. Keep in mind that whenever applying multiplication or division, the sizes of the operands must be compatible with each other in terms of matrix operations. Whenever two operands are not compatible in their sizes, an error saying that the arguments of the calculation are non-conformant will be invoked. The easiest way to check prior to performing calculations if the shapes of variables are conformant is by calling the **`size`** method over them. It can be the case that we want to perform an **elementwise calculation** over matrices. We shall indicate our intention with the **dot (.) operator**, e.g., elementwise multiplication (instead of matrix multiplication) between matrices A and B is denoted by $A.*B$ (instead of $A*B$).

As for indexing in Octave, a somewhat uncommon indexing convention is employed as elements are indexed starting with 1 (as

CODE SNIPPET

```
octave:16> M=reshape(1:10,2,5)
M =

 1   3   5   7   9
 2   4   6   8  10

octave:17> M(:,[2,4,5])
ans =

 3   7   9
 4   8  10

octave:18> M(:,[5,2,4])
ans =

  9   3   7
 10   4   8
```

Figure 2.7: An illustration of indexing matrices in Octave.

opposed to 0 which is frequently encountered in most other programming languages). Figure 2.7 includes several examples of how indexing can be used to select certain elements from a matrix.

Figure 2.7 reveals a further useful property of Octave. Even if the result of a computation does not get assigned to a variable (or multiple variables) using the $=$ operator, the result of the lastly performed computation is automatically saved into an artificial variable, named **`ans`** (being a shorthand for answer). This variable acts as any other user created variable, so invoking something like `2*ans` is totally fine in Octave. Note however, that any upcoming command invoked in Octave will override the value stored in the artificial variable `ans`, so in case we want to reutilize the results of a certain calculation, then we should definitely assign that result to a dedicated variable for future use.

2.1.5 *Broadcasting*

Recall the function from Figure 2.5. At that point we argued that arguments must match their sizes in order arithmetic operations to make sense. That is, we can add and subtract two 5×6 matrices if we want, however, we are unable to perform such an operation between a 5×6 and a 5×7 matrices for instance.

What Octave does in such situations is that it automatically attempts to apply **broadcasting**. This means that Octave tries to make

CODE SNIPPET

```
%% generate 10000 100-dimensional observations
X=rand(10000, 100);

tic
mu=mean(X);
X_centered=X;
for l=1:size(X,1)
    X_centered(l,:) -= mu;
end
toc

>>Elapsed time is 0.153826 seconds.
```

Figure 2.8: Centering the dataset with `for` loop

the most sense out of operations that are otherwise incompatible from an algebraic point of view due to a mismatch in the sizes of their operands. Octave will certainly struggle with the previous example of trying to sum $5x6$ and $5x7$ matrices, however, there will be situations when it will provide us with a result even though our calculation cannot be performed from a linear algebraic point of view.

For instance, given some matrix $A \in \mathbb{R}^{n \times m}$ and a (column) vector $\mathbf{b} \in \mathbb{R}^{m \times 1}$, the expression $A - \mathbf{b}^\intercal$ is treated as if we were to calculate $A - \mathbf{1}\mathbf{b}^\intercal$, with $\mathbf{1}$ denoting a vector in $\mathbb{R}^{n \times 1}$ full of ones, i.e. $\begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$. The outer product $\mathbf{1}\mathbf{b}^\intercal$ simply results in a matrix which contains the (row) vector $\mathbf{b}^\intercal$ in all of its rows and which has the same number of rows as matrix A.

CODE SNIPPET

```
tic
X_centered_broadcasted=X-mean(X);
toc

>>Elapsed time is 0.013844 seconds.
```

Figure 2.9: Centering the dataset relying on broadcasting

Figure 2.9 illustrates broadcasting in action when we subtract the row vector that we get by calculating the column-wise mean values of matrix X and subtracting this row vector from *every row* of matrix X in order to obtain a matrix which has zero expected value across all of its columns.

? For an arbitrarily sized matrix A, what would be the effect of the command `A-3`? How would you write it down with standard linear algebra notation?

2.1.6 *Vectorization*

Vectorization is the process when we try to express some sequential computation in the form of matrix operations. The reason to do so is that this way we can observe substantial speedup, especially if our computer enjoys access to some highly-optimized, hence extremely fast matrix libraries, such as **BLAS** (Basic Linear Algebra Subprograms), **LAPACK** (Linear Algebra PACKage), **ATLAS** (Automatically Tuned Linear Algebra Software) or **Intel MKL** (Math Kernel Library). Luckily, Octave can build on top of such packages, which makes it really efficient when it comes to matrix calculations.

Suppose we are interested in the columnwise sums of some matrix $X \in \mathbb{R}^{m \times n}$. That is, we would like to know $s_l = \sum_{k=1}^{m} x_{kl}$ for every $1 \leq l \leq n$. A straightforward implementation can be found in Figure 2.10, i.e., where we write two nested for loops to iterate over all the x_{kl} elements of matrix X and for each index while incrementing the appropriate cumulative counter s_l.

CODE SNIPPET

```
X=rand(10000, 100);
tic
col_sum = zeros(1, size(X, 2));
for k=1:size(X,1)
  for l=1:size(X,2)
    col_sum(l) += X(k,l);
  endfor
endfor
toc

>>Elapsed time is 9.45435 seconds.
```

Figure 2.10: Calculating the columnwise sum of a matrix with a `for` loop.

We can, however, observe that matrix multiplications are also inherently applicable to express summations. This simply follows from the very definition of matrix multiplication, i.e., if we define $Z = XY$ for any two matrices $X \in \mathbb{R}^{m \times n}$ and $Y \in \mathbb{R}^{n \times p}$, we have $z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj}$. The code snippet in Figure 2.11 utilizes exactly this kind of idea upon speeding up substantially the calculation of the columnwise sums of our matrix by left multiplying it with a vector of all ones. By comparing the reported running times of the two implementations, we can see that the vectorized implementation has a more than 5000-fold speedup compared to the one which explicitly uses for loops during the calculation even in the case of a moderately sized 10000×100 matrix.

CODE SNIPPET

```
tic
vectorized_col_sum = ones(1, size(X,1)) * X;
toc

>>Elapsed time is 0.00158882 seconds.
```

Figure 2.11: Calculating the columnwise sum of a matrix in a vectorized fashion *without* a `for` loop.

Note that we could have come up with a 'semi-vectorized' implementation for calculating the columnwise sum of our matrix as illustrated in Figure 2.12. In this case we are simply making use of the fact that summing up the row vectors in our matrix also provides us with the columnwise sums of our matrix. This way we can manage to remove one of the unnecessary for loops, but we are still left with one of them. This in-between solution for eliminating needless loops from our implementation gives us a medium speedup, i.e., this versions runs nearly 60-times slower than the fully vectorized one, however, it is still more than 100-times faster compared to the non-vectorized version.

CODE SNIPPET

```
tic
semi_vectorized_col_sum=zeros(1,size(X,2));
for k=1:size(X,1)
  semi_vectorized_col_sum += X(k,:);
end
toc
>>Elapsed time is 0.091974 seconds.
```

Figure 2.12: Calculating the columnwise sum of a matrix in a semi-vectorized fashion with single a `for` loop.

As another example, take the code snippets in Figure 2.13 and Figure 2.14, both of which transforms a 10000×100 matrix such that after the transformation every row vector has a unit norm. Similar to the previous example, we have a straightforward implementation using a for loop (Figure 2.13) and a vectorized one (Figure 2.14). Just as before, running times are reasonably shorter in the case of the vectorized implementation.

The main take-away is that vectorized computation in general not only provides a more concise implementation, but one which is orders of magnitude faster compared to the non-vectorized solutions. If we want to write efficient code (and most often – if not always – we do want), then it is crucial to identify those parts of our computation which can be expressed in a vectorized manner.

CODE SNIPPET

```
X=rand(10000, 100);
tic
X_unit=X;
for l=1:size(X,1)
  X_unit(l,:) /= norm(X(l,:));
end
toc

>>Elapsed time is 0.283947 seconds.
```

Figure 2.13: Transforming vectors in a matrix to unit-norm by relying on a `for` loop.

CODE SNIPPET

```
tic
norms=sqrt(sum(X.*X, 2));
X_unit_vectorized=X./norms;
toc

>>Elapsed time is 0.0134549 seconds.
```

Figure 2.14: Transforming vectors in a matrix to unit-norm in a vectorized fashion *without* a `for` loop.

Writing a vectorized implementation might take a bit more effort and time compared to its non-vectorized counterpart if we are not used to it in the beginning, however, do not hesitate to spend the extra time cranking the math, as it will surely pay off in the running time of our implementation.

? Can you think of a vectorized way to calculate the covariance matrix of some sample matrix X? In case the concept of a covariance matrix sounds unfamiliar at the moment, it can be a good idea to look at Figure 3.5.

2.2 *Plotting*

There is a wide variety of plotting functionalities in Octave. The simplest of all is the **`plot`** function, which allows us to create simple x-y plots with linear axes. It takes two vectors of the same length as input and draws a curve based on the corresponding indices in the two vector arguments. Figure 2.15 illustrates the usage of the function with its output being included in Figure 2.16.

CODE SNIPPET

```
x=-3:.01:4;  % create a vector in [-3,4] with a step size 0.1
plot(x, 2*x.^3)
xlabel('x')
ylabel("f(x)=2x^3")
```

Figure 2.15: Code for drawing a simple function $f(x) = 2x^3$ over the range $[-3, 4]$.

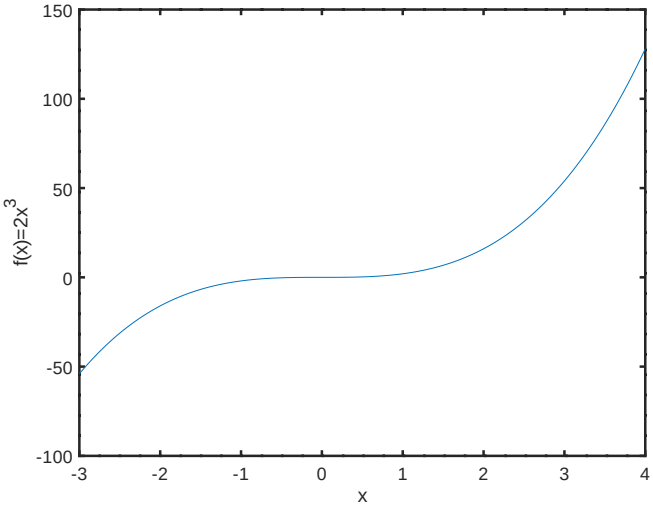


Figure 2.16: The resulting output of the code snippet in Figure 2.15 for plotting the function $f(x) = 2x^3$ over the interval $[-3, 4]$.

For the demonstration of further plotting utilities of Octave, consider the tiny example dataset found in Table 2.1 containing height and weight information of 8 people. Prior to running any core data mining algorithm on a dataset like that, it is often a good idea to familiarize with the data first. Upon getting to know the data better, one typically checks out how the observations are distributed.

Possibly the simplest way to visualize the (joint) empirical distribution for a pair of random variables (or features in other words). One can visualize the empirical distribution of the feature values by creating a so called **scatter plot** based on the observed feature values. The Octave code and its respective output are included in Figure 2.17 and Figure 2.18, respectively.

By applying a scatter plot, one inherently limits himself/herself to focus on a pair of random variables, which is impractical for truly high-dimensional data that we typically deal with. Dimensionality reduction techniques, to be discussed in more detail later in Chapter 6 are possible techniques to aid the visualization of datasets with high number of dimensions.

height (cm)	weight (kg)
187	92
172	77
200	101
184	92
162	70
176	81
172	68
166	55

Table 2.1: Example dataset containing height and weight measurements of people.

CODE SNIPPET

```
scatter(D(:,1), D(:,2))
xlabel('height (cm)')
ylabel('weight (kg)')
```

Figure 2.17: Code for drawing a scatter plot of the heights of the people from the sample to be found in Table 2.1.

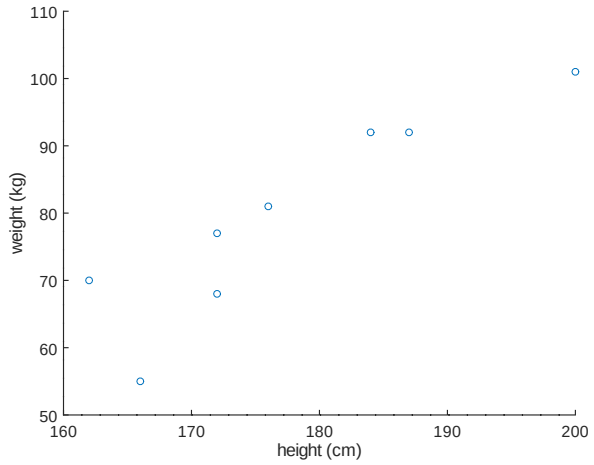


Figure 2.18: Scatter plot of the heights and weights of the people from the sample to be found in Table 2.1.

The other frequently applied visualization form is for histograms, which aim to approximate some distribution by drawing an empirical frequency plot for the observed values. The way it works is that it divides the span of the random variable into a fixed number of equal bins and count the number of individual observations that fall within a particular range. This way, the more observations fall into a specific range, the higher bar is going to be displayed for that. A visual illustration of it and the corresponding Octave code which created it can be found in Figure 2.19 and Figure 2.20, respectively.

CODE SNIPPET

```
hist(D(:,1),4)
xlabel('height (cm)')
ylabel('frequency')
```

Figure 2.19: Code for drawing a histogram of the heights of the people from the sample to be found in Table 2.1.

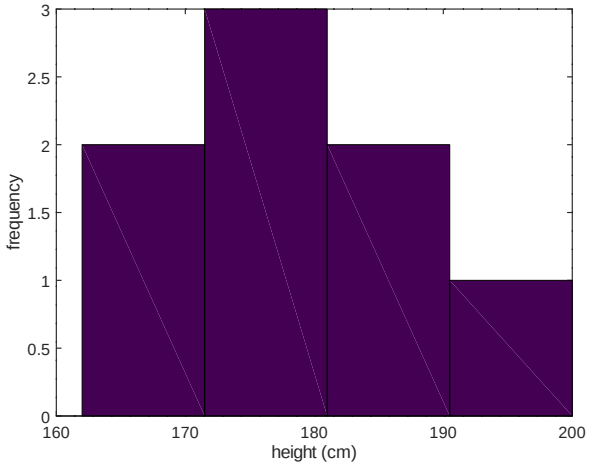


Figure 2.20: Histogram of the heights of the people from the sample to be found in Table 2.1.

2.3 *A brief comparison of Octave and numpy*

In the recent years, **Python** has gained a fair amount of popularity as well. Python is similar to Octave in that it is also a script language. However, Python is intended to be a more general purpose programming language compared to Octave and it was not primarily designed for numerical computations. Thanks to the huge number of actively maintained libraries for Python, it is now also possible to perform matrix computations with the help of **numpy** and **scipy** packages (among many others).

It also has to offer similar functionalities to that of Octave in terms of data manipulation, including matrix calculations. In order to obtain the same core functionalities of Octave, which is of primarily interest for this book, one need to install and get to know a handful of non-standard Python packages, all with its own idiosyncrasy. For this reason, we will use Octave code throughout the book for illustrative purposes.

Since there are relatively few major differences, someone fairly familiar with Octave can relatively quickly start writing Python code related to matrices. Perhaps the most significant and error-prone difference in the syntax of the two languages is that the $*$ operator denotes regular matrix multiplication in Octave (and its relatives such as Matlab), whereas the same operator when invoked for numpy multidimensional arrays acts as elementwise multiplication. The latter is often referenced as the **Hadamard product** and denoted by $\circ$.

Recall that in Octave, whenever one wants to perform some elementwise operation, it can be expressed with an additional dot (.) symbol as already mentioned in Section 2.1.4. For instance, $.*$ denotes elementwise multiplication in Octave. In order to see the differ-

CODE SNIPPET

```
X = [1 -2; 3 4];
Y = [-2 2; 0 1];
X*Y   % perform matrix multiplication
>>ans =
  -2    0
  -6   10

X.*Y    % perform elementwise multiplication
>>ans =
  -2  -4
   0   4
```

Figure 2.21: Comparing matrix-style and elementwise multiplication of matrices.

ence between the two kinds of multiplication in action, see the code snippet in Figure 2.21.

2.4 *Summary of the chapter*

Readers of this chapter are expected to develop familiarity with numerical computing, including the basic syntax of the language and also its idiosyncracies related to indexing, broadcasting and vectorization which allows us to write efficient code thanks to the highly optimized and parallelized implementation of the linear algebra packages Octave relies on in the background.

3 | Basic concepts

Learning Objectives:

- Goal of data mining
- Bonferroni principle
- Simpson's paradox
- Data preprocessing techniques
- Basic concepts from information theory

In this chapter we overview the main goal of data mining. By the end of the chapter, readers are expected to

- develop an understanding on the ways datasets can be represented,
- show a general sensitivity towards the ethical issues of data mining applications,
- be able to distinguish the different measurement scales of random variables and illustrate them,
- produce and efficiently implement summary statistics for the random variables included in datasets,
- get familiar with basic concepts of information theory,
- understand the importance, apply and implement various data manipulation techniques.

Data mining is a branch of computer science which aims at providing efficient algorithms that are capable of extracting useful knowledge from vast amounts of data. As an enormous amount of data gets accumulated by the everyday activities, such as watching videos on YouTube, ordering products via Amazon, sending text messages on Twitter. As interacting with our environment gets tracked in an ever-increasing pace even for our most basic every day activities, there are a variety of ways in which data mining algorithms can influence our lives.

Needless to say, there exists enormous application possibilities in applying the techniques of data mining with huge economic potential as well. As a consequence data has often been coined as the new oil[1], due to the fact that having access to data these days can generate such an economic potential to businesses as fossil energy carrier used to do so. Handling the scale of data that is generated raises many

[1] https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data

interesting questions both in terms of scientific and engineering nature. In the following, we provide a few examples illustrating the incredible amount of data that is continuously being accumulated.

As an illustration regarding the abundance of data one can think of the social media platform Twitter. On an average day Twitter users compose approximately half a billion short text messages, meaning that nearly 6,000 tweets are posted every single second on average. Obviously, tweets are not uniformly distributed over time, users being more active during certain parts of the day and particular events triggering enormous willingness to write a post. As an interesting trivia, the highest tweet-per-second (TPS) rate ever recorded up to 2018 dates back to August 2013, when 143,199 TPS was recorded[2]. Needless to say, analyzing this enormous stream of data can open up a bunch of interesting research questions one can investigate using data mining and knowledge discovery [3].

As another example relating to textual data, Google revealed that its machine translation service is fed 143 billion words every single day[4] as of 2018. To put this amount of text into context – according to Wikipedia[5] – this roughly corresponds to 75,000 copies of the world's longest novel which consists of 10 volumes and nearly 2 million words.

As an even more profound example, one can think of the experiments conducted at the **Large Hadron Collider** (LHC) operated by **CERN**, the European Organization for Particle Physics. There are approximately 1 billion particle collisions registered in every second which yield an enormous amount of 1 Petabyte (10^{15} bytes) of data. As this amount of data would be unreasonable and wasteful to store directly, the CERN Data Centre distills it first so that 1 Petabyte of data to archive accumulates over a period of 2–3 *days* (instead of a single second) as the end of 2017. As a consequence 12.3 Petabyte of data was archived to magnetic tapes during October 2017 alone and the total amount of permanently archived data surpassed 200 Petabytes in the same year[6].

[2] https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html

[3] Morstatter et al. 2013, Gligoric et al. 2018

[4] https://www.businessinsider.de/sundar-pichai-google-translate-143-billion-words-daily-2018-7

[5] https://en.wikipedia.org/wiki/List_of_longest_novels#List

[6] https://home.cern/about/updates/2017/12/breaking-data-records-bit-bit

3.1 *Goal of data mining*

Most real world observations can be naturally thought of as some (high-dimensional) vectors. One can imagine for instance each customer of a web shop as a vector in which each dimension of the vector is assigned to some product and the quantity along a particular dimension indicates the number of items that user has purchased so far from the corresponding product. In the example above, the possible outcome of a data mining algorithm could group customers with similar product preferences together.

The general goal of data mining is to come up with previously unknown valid and relevant knowledge from large amounts of datasets. For this reason data mining and knowledge discovery is sometimes referred as synonyms, the schematic overview of which is summarized in Figure 3.1.

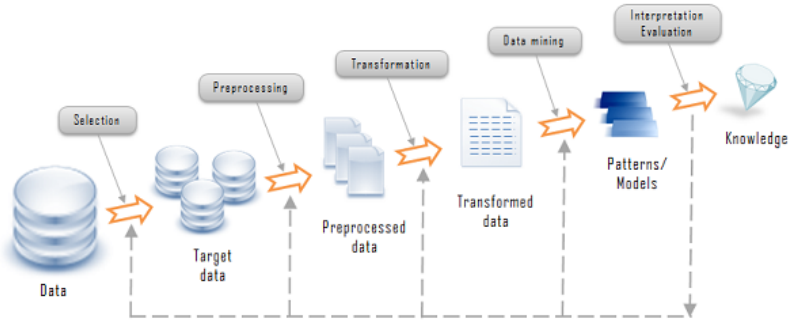


Figure 3.1: The process of knowledge discovery

3.1.1 *Correlation does not mean causality*

A common fallacy when dealing with datasets is to use correlation and causality interchangeably. When a certain phenomenon is caused by another, it is natural to see high correlation between the random variables describing the phenomena being in a causal relationship. This statement is not necessarily true in the other direction, i.e., just because it is possible to notice a high correlation between two random variables it need not follow that any of the events have a causal effect on the other one.

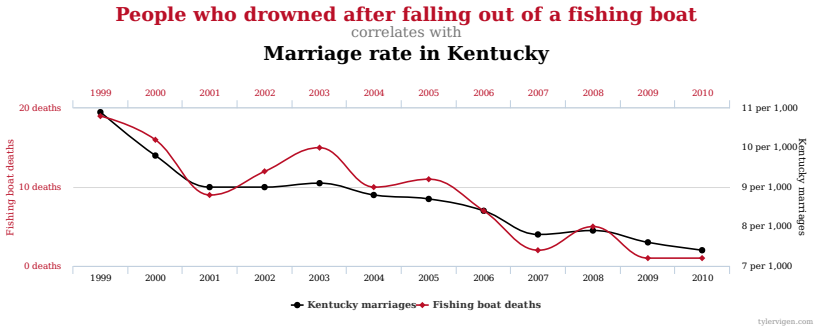


Figure 3.2: Example of a spurious correlation. Original source of the plot: http://www.tylervigen.com/spurious-correlations

Figure 3.2 shows such a case when high correlation between two random variables (i.e., the number of people drowned and the marriage rate in Kentucky) is very unlikely to be in causal relationship with each other.

3.1.2 *Simpson's paradox*

Simpson's paradox also reminds us that we shall be cautious when drawing conclusions from some dataset. As an illustration for the paradox, inspect the hypothetical admittance rates of an imaginary university as included in Table 3.1. The admittance statistics are broken down with respect the gender of applicants and the major that they applied for.

	Applied/Admitted	
	Female	Male
Major A	7/100	3/50
Major B	91/100	172/200
Total	98/200	175/250

Table 3.1: Example dataset illustrating Simpson's paradox

At first glance, it seems that females have a harder time getting admitted to the university overall, as their success rate is only 49% (98 admitted out of 200), whereas males seem to get admitted more seamlessly with a 70% admittance rate (175 out of 200). Looking at these marginalized admittance rates, decision makers of this university might suspect that there might be some unfair advantage given to male applicants.

Looking at the admittance rates broken down to each major on the other hand, shows us a seemingly contradictory pattern. For Major A and B female applicants show a 7% and 91% success rate, respectively, compared to the 6% and 86% success rate for males. So, somewhat counter-intuitively, females have a higher acceptance rate than males on both major A and B, yet their aggregated success rate falls behind that of males. Before reading onwards, can you come up with an explanation for this mystery?

In order to understand what is causing this phenomenon, we have to observe that females and males have a different tendency towards applying for the different majors. Females tend to apply in an even fashion as there are 100–100 applicants for both Major A and B, however, for the males there is a preference towards Major B, which seems to be an easier way to go for in general. Irrespective of the gender of the applicant, someone who applied to Major B was admitted with 87.7% chance, i.e., (91+172)/(100+200), whereas the gender-agnostic success rate is only 6.7% for Major A, i.e., only 10 people out of the 150 applicants were admitted for that major.

The **law of total probability** tells us how to come up with probabilities that are 'agnostic' towards some random variable. In a more rigorous mathematical language, defining some observation-agnostic probability is called marginalization and the probability we obtain as

marginal probability.

Let us define random variable M as a person's choice for a major to enroll to and G as the gender a person. Formally, given these two random variables, the probability for observing a particular realization m for variable M can be expressed as a sum of joint probabilities over all the possible values g that random variable G can take on. Formally, $P(M = m) = \sum_{g \in G} P(M = m, G = g)$.

Another concept we need to be familiar with is the **conditional probability** of some event $M = m$ given $G = g$. This is defined as $P(M = m|G = g) = \frac{P(M=m,G=g)}{P(G=g)}$, i.e., the fraction of the joint probability of the two events divided by the marginal probability of the event on which we wish to condition on. If we introduce another random variable S which indicates if an application is successful or not, we can express the following equalities by relying on the definition of the conditional probability

$$
\begin{aligned}
&P(S = s|M = m, G = g) \cdot P(M = m|G = g) = \\
&= \frac{P(S = s, M = m, G = g)}{P(M = m, G = g)} \cdot \frac{P(M = m, G = g)}{P(G = g)} = \\
&= \frac{P(S = s, M = m, G = g)}{P(G = g)} = \\
&= P(S = s, M = m|G = g).
\end{aligned}
$$

This means that the probability of a person being successfully admitted to a particular major **given** his/her gender can be decomposed into the product of two conditional probabilities, i.e.,

1. the probability of being successfully admitted **given** the major he/she applied for and his/her gender and
2. the probability of applying for a particular major **given** his/her gender.

Recalling the law of total probability, we can define probability that someone is successfully admitted **given** his/her gender as

$$
\begin{aligned}
P(S = s|G = g) &= \sum_{m \in \{A,B\}} P(S = s, M = m|G = g) = \\
&= \sum_{m \in \{A,B\}} P(S = s|M = m, G = g) \cdot P(M = m|G = g).
\end{aligned}
$$

Based on that, the admittance probability that for females emerges as

$$P(S = success|G = female) = \frac{7}{100} \cdot \frac{100}{200} + \frac{100}{200} \cdot \frac{91}{100} = \frac{98}{200} = 0.49,$$

whereas that for males is

$$P(S = success|G = male) = \frac{3}{50} \cdot \frac{50}{250} + \frac{172}{200} \cdot \frac{200}{250} = \frac{175}{250} = 0.7.$$

This break-down for the probability of the success for the different genders unveils that the probability that we observe in the major-agnostic case can deviate from the probability for success that we get for the major-aware case. The reason for the discrepancy was due to the fact that females had an increased tendency for applying to the major which was more difficult to get in. Once we look at the admittance rates for the two majors separately, we can see, that – contrarily to our first impression based on the aggregated data – female applicants were more successful during their applications.

3.1.3 *Bonferroni's principle*

Bonferroni's principle reminds us that if we repeat some experiment multiple times, we can easily observe some phenomenon to occur frequently purely originating from our vast amount of data that we are analyzing. As a consequence, it is of great importance that whenever we notice some seemingly interesting pattern of high frequency, to be aware of the number of times we were about to observe that given pattern purely due to chance. This way we can mitigate the effects of creating false positive alarms, i.e., claiming that we managed to find something interesting when this is not the case in reality.

Let us suppose that all the people working at some imaginary factory are asocial at their workspace, meaning that they are absolutely uninterested in becoming friends with their co-workers. The manager (probably unaware of the employees not willing to make friendships) decides to collect pairs of workers who could become soul mates. The manager defines potential soul mates as those pair of people who order the exact same dishes during lunchtime at the factory canteen where the workers can choose between s soups, m main dishes and d many desserts. For simplicity let us assume that the canteen serves $s = m = d = 6$ kind of meals each for $w = 1,200$ workers and the experiment runs over one week. The question is, how many potential soul mates would we find under these conditions by chance?

Let us assume that the employees do not have any dietary restrictions and have no preference among the dishes the canteen can offer, for which reason they select their lunch every day randomly. This means that there are $s \cdot m \cdot d = 6^3 = 216$ many equally likely different combinations of lunches for which reason, the probability of a worker choosing a particular lunch configuration (a soup, main course, dessert triplet) is $6^{-3} \approx 0.0046$.

It turns out that the probability of two workers choosing the same lunch configurations independently have the exact same probability as one worker going for a particular lunch. Since every employee has 216 options for the lunch, the number of different ways a pair

of people can arrange their lunch is 216^2. As any of the 216 possible lunch configurations can be the one they match on, the probability of two workers ordering the same dishes is again $\frac{216}{216^2} = 216^{-1} \approx 0.0046$. This means that out of 1,000 pairs of people, we would expect to see less than 5 cases when the same meals are ordered.

This seems like a tolerable amount of false positive alarms which is produced by people simply behaving by chance. However, if we add that the fact that there are $w = 1200$ people employed in the factory, we immediately find a much higher number of erroneously identified soul mates. The 1200 employees form $\binom{1200}{2} = \frac{1200*1199}{2} = 719,400$ pairs of workers, hence the number of unjustifiable soul mates we identify per day amounts to 3,300 per a day. Which is above 21,000 false matches over the period of one week (disregarding the fact that over the course of multiple days, we would certainly identify certain pairs of people more than once, so we would register less than 21,000 unique cases).

3.1.4 *Ethical issues of data mining*

Since data mining algorithms affect our every day lives in numerous ways, it is of utmost importance to strive for designing such algorithms that are as fair as possible, e.g., they do not privilege or disadvantage certain individuals based on their gender or nationality even in an implicit manner. [7]

As the decisions made or augmented by data mining algorithms are ubiquitous and often high-impact, it is crucial to create as accountable and transparent algorithms as possible. By carefully selecting the input for the data mining algorithms can go a long way. Imagine that a company wants to aid its recruiting procedure by relying on a data mining solution which gives recommendation on the expected success of the candidates during the interview based on historic data. Arguably, the gender of an applicant should be independent from his or her merits and qualifications. As such it makes sense to not to feed such an algorithm with the gender of the applicants as input. Yet another solution is to provide the algorithm an even proportion of successful and unsuccessful applicants from each gender, in order to minimize the chances for a preference towards any gender to be developed by the algorithm. Additionally, one can incorporate additional soft or hard constraints into any algorithm, so that they behave in a more adequate and ethical manner.

[7] https://arstechnica.com/information-technology/2016/02/the-nsas-skynet-program-may-be-killing-thousands-of-innocent-people/

? Can you think of real word use cases of data mining problems where ethical issues can arise?

3.2 *Representing data*

The most convenient way to think of the datasets that the majority of data mining algorithms operate upon is the tabular view. In this analogy the problem at hand can be treated as (a potentially gigantic) spreadsheet with several rows – corresponding to data objects – and columns, each of which includes observed attributes with respect the different aspects of these data objects.

Different people tend to think of this gigantic spreadsheet differently, hence different naming conventions coexist among practitioners which are listed in Table 3.2.

Data object	Data attribute
record	field
data point	dimension
sample/measurement	variable
instance/sample	attribute, feature

Table 3.2: Typical naming conventions for the rows and columns of datasets.

Another important aspect of the datasets we work with is the measurement scale of the individual columns in the data matrix (each corresponding to a random variable). A concise summary of the different measurement scales and some of the most prototypical statistics which can be calculated for them is included in Table 3.3.

Type of attribute		Description	Examples	Statistics
Categorical	Nominal	Variables can be checked for equality only	names of cities, hair color	mode, entropy, correlation, χ^2-test
	Ordinal	> relation can be interpreted among variables	grades, {fail, pass, excellent}	median, percentiles
Numerical	Interval	The difference of two variables can be formed and interpreted	shoe sizes, dates, °C	mean, deviation, significance (e.g. F,t-) tests
	Ratio	Ratios can be formed from values of the variables of this kind	age, length, temperature in Kelvin	percent, geometric/harmonic mean, variation

Table 3.3: Overview of the different measurement scales.

3.2.1 *Data transformations*

It is often a good idea to perform some preprocessing steps on the raw data we have access to. This means that we perform some transformation over the data matrix, either in a column or a row-oriented manner.

3.2.2 *Transforming categorical observations*

First of all, as we would like to treat observations as vectors (a sequence of scalars), we should find a way to transform nominal observations into numeric values. As an example, let us assume that for a certain data mining problem we have to make a decision about users based on their age and nationality, e.g. a data instance might look like `(32, Brazilian)` or `(17, Belgian)`. Here we cover some of the most commonly used techniques for turning nominal feature values into numerical ones.

In our case, nationality is a categorical – more precisely a nominal – variable. One option is to simply deterministically map each distinct value of the given feature a separate integer which then identifies it and simply replace them consistently based on this mapping. Table 3.4 (b) contains such a transformation for the data from Table 3.4 (a). While we can certainly obtain scalar vectors that way, this is not the best idea since, this would suggest that there exists an ordering between different nationalities, which arguably is not the case. Hence, alternative encoding mechanisms are employed most often.

Encoding categorical values with the one-hot-encoding schema is a viable technique, in which an n-ary categorical variable, that is a categorical variable with n distinct feature values, is split into n different binary features. This way we basically map the exact categorical value into a vector of dimensions n, which has exactly one position at which a value 1 is stored, indicative of the value taken by the variable, and has zeros in all other positions. This kind of transformation is illustrated in Table 3.4 (c).

Encoding an n-ary categorical variable with n distinct binary features carries the possible danger of falling into the so called **dummy variable trap** which happens when you can infer the value of a random variable without actually seeing it. When n distinct variables are created for an n-ary variable, this is exactly the case as observing $n-1$ out of these newly created variables, one can tell with absolute certainty the exact value for the remaining n^{th} variable. This phenomena is coined as the phenomenon of **multicollinearity** which causes the dummy variable trap. In order to avoid it, a common technique is to simply drop one of the n binary features, hence getting rid of the problem of multicollinearity. Table 3.4 (d) illustrates the solution

ID	Age	Nationality
1	23	Brazilian
2	55	Belgian
3	31	Brazilian
4	72	Korean
⋮	⋮	⋮

(a) Sample data with categorical variable (Nationality) prior to transformation

ID	Age	Nationality
1	23	1
2	55	2
3	31	1
4	72	3
⋮	⋮	⋮

(b) Sample data with categorical variable (Nationality) mapped to numeric values

ID	Age	BEL	BRA	KOR
1	23	0	1	0
2	55	1	0	0
3	31	0	1	0
4	72	0	0	1
⋮	⋮	⋮	⋮	⋮

(c) Sample data with categorical variable (Nationality) transformed as one-hot encoding

ID	Age	BEL	BRA
1	23	0	1
2	55	1	0
3	31	0	1
4	72	0	0
⋮	⋮	⋮	⋮

(d) Sample data with categorical variable (Nationality) transformed as reduced dummy variable

Table 3.4: Illustration of the possible treatment of a nominal attribute. The newly introduced capitalized columns correspond to binary features indicating whether the given instance belongs to the given nationality, e.g., whenever the BRA variable is 1, the given object is Brazilian.

of applying a reduced set of dummy variables after simply dropping one of the binary random variables for one of the nationalities.

Introducing new features proportional to the number of distinct values some categorical variable can take, however, might carry another potential problem, i.e., this way we can easily experience an enormous growth in the dimensionality for the representation of our data points. This can be dangerous for which reason it is often the case that we do not introduce a separate binary feature for every possible outcome of a categorical variable, however, bin them into groups of feature values and introduce a new meta-feature for every bin instead, decreasing the number of newly created features that way. More sophisticated schemas can be thought of, however, it turns out that hashing feature values, i.e. mapping n different outcome variables to $m \ll n$ distinct ones using a hash function, can produce surprisingly good results in large scale data mining and machine learning applications [8] with theoretical guarantees [9].

[8] Weinberger et al. 2009

[9] Freksen et al. 2018

3.2.3 *Transforming numerical observations*

Once we have numerical features, one of the simplest and most frequently performed data transformation is **mean centering data**. Mean centering involves making all of the variables in your dataset behave such that they have an expected value of zero. This way the

? Can you recall why does handling the nominal attribute in the example dataset in Figure 3.4 (a) makes more sense as illustrated in 3.4 (c) (splitting) as opposed to the strategy presented in 3.4 (b) (mapping)? Can you list situations when the strategy in 3.4 (b) is less problematic?

CODE SNIPPET

```
pkg load statistics
M = mvnrnd([6 -4], [6 1; 1 .6], 50);
Mc=M-mean(M);                    % mean centering the data
Ms=Mc./std(Mc);                  % standardize the data
L=chol(inv(cov(Ms)))';
Mw=Ms*L;                         % whitening the data
Mm=(M-min(M))./(max(M)-min(M)); % min-max normalize the data
Mu=Mc./sqrt(sum(Mc.^2,2));       % unit normalize the data
```

Figure 3.3: Various preprocessing steps of a bivariate dataset

transformed values represent the extent to which they differ from the prototypical mean observation. In order to perform the transformation, one has to calculate μ, being the mean of the untransformed random variable, and subtract this quantity from every observation of the corresponding random variable. The Octave code performing this kind of transformation and its corresponding geometrical effect is included in Figure 3.3 and Figure 3.4 (b), respectively.

Upon **standardizing** a (possibly multivariate) random variable X, we first subtract $\boldsymbol{\mu}$, the empirical mean from all the observations. Note that this step is identical to mean centering up to this point. As a subsequent step, we then need to rescale the random variable by the variable–wise standard deviation σ.

By doing so, we express observations as **z–score**s, which tells us the extend to which a particular observation differs from its typically observed value, i.e., its mean expressed in units of the standard deviation. The geometric effects of standardizing a bivariate variable is depicted in Figure 3.4 (c) and the corresponding Octave code is found in Figure 3.3.

Example 3.1. *For simplicity, let's deal with a univariate random variable in this first example, H for the height of people. Suppose we have a sample of* $X = \{75, 82, 97, 110, 46\}$ *thus having*

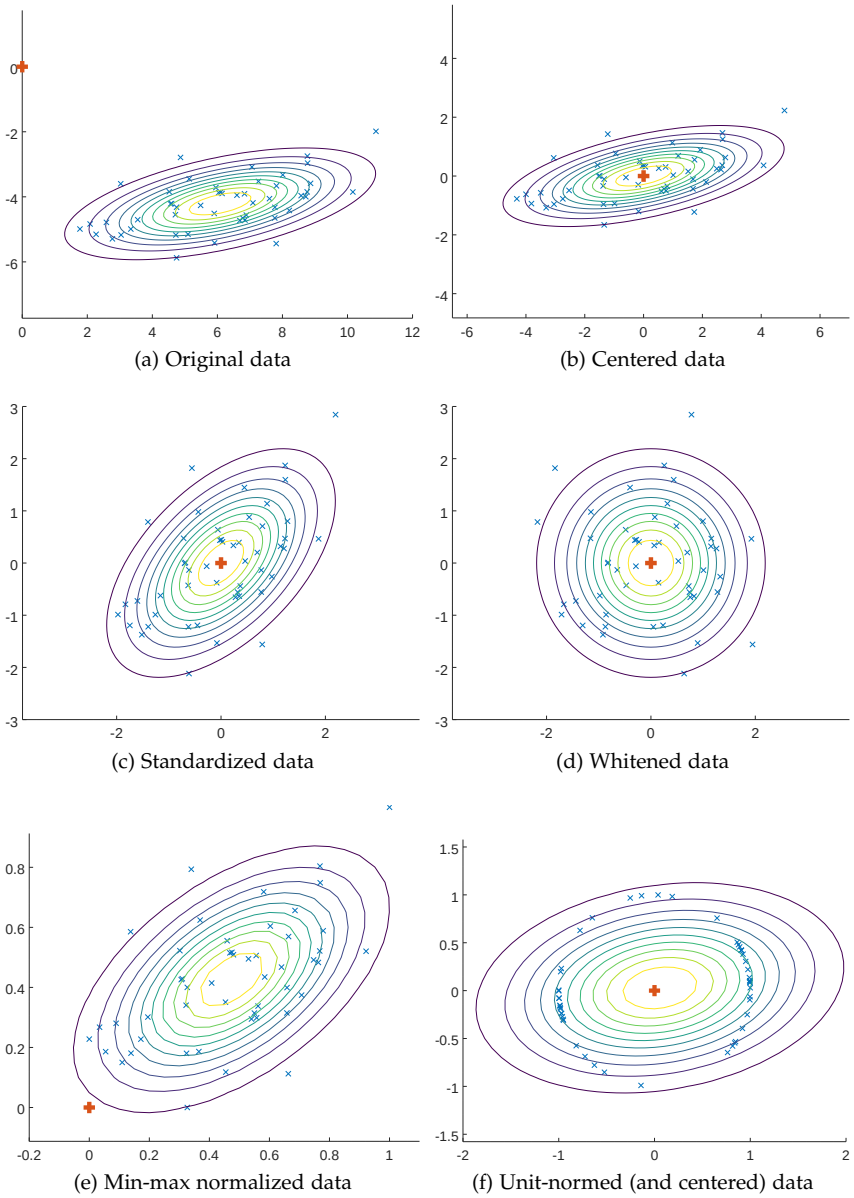
$$\mu = \frac{75 + 82 + 97 + 110 + 46}{5} = \frac{410}{5} = 82.$$

The standard deviation of the sample is defined by the quantity

$$s = \sqrt{\frac{\sum_{i=1}^{n}(X_i - \mu)^2}{n-1}}.$$

For the given sample, we thus have

$$s = \sqrt{\frac{(75-82)^2 + (82-82)^2 + \ldots + (46-82)^2}{5-1}} \approx 24.56$$



(a) Original data

(b) Centered data

(c) Standardized data

(d) Whitened data

(e) Min-max normalized data

(f) Unit-normed (and centered) data

Figure 3.4: Various transformations of the originally not origin centered and correlated dataset (with the thick orange cross as the origin).

The process of eliminating the correlation from the data is also called **whitening**. The process of whitening transforms a set of data points with an arbitrary covariance structure into such set of points that their **covariance matrix** becomes the identity matrix, i.e., the individual dimensions have a variance of one uniformly, and pairwise covariances between pairs of distinct dimensions become zero.

Now suppose that we have a matrix $X \in \mathbb{R}^{n \times m}$ with a covariance matrix $C \in \mathbb{R}^{m \times m}$. Without loss of generality, we can assume that X is already mean centered. Assuming so means that $X^\intercal X \propto C$, i.e., the result of the matrix product $X^\intercal X$ is directly proportional to the empirical covariance matrix calculated over our set of observations. Indeed, if we divide $X^\intercal X$ by the number of observations n (or $n-1$), we would exactly obtain the biased (unbiased) estimation for the covariance matrix.

Now the question is what transformation $L \in \mathbb{R}^{m \times m}$ do we have to apply over X so that the covariance matrix we obtain for the transformed dataset XL equals the identity matrix? An identity matrix (denoted by I) is such a matrix which has non–zero elements only in its main diagonal and those non–zero elements are uniformly ones.

What this means is that initially, we have $X^\intercal X \propto C$, and we are searching for some linear transformation L, such that $(XL)^\intercal (XL) \propto I$ is the case. Let us see, how is this possible.

$$
\begin{aligned}
(XL)^\intercal (XL) &= (L^\intercal X^\intercal)(XL) && \triangleright \text{because } (AB)^\intercal = B^\intercal A^\intercal \forall A, B \\
&= L^\intercal (X^\intercal X) L && \triangleright \text{by associativity of matrix multiplication} \\
&= L^\intercal C L. && \triangleright \text{by our assumption}
\end{aligned}
$$

This means that the linear transformation L, which makes our data matrix X decorrelated has to be such that $L^\intercal C L = I$. This also means that $LL^\intercal = C^{-1}$ – with C^{-1} denoting the inverse of matrix C. The latter observation is derived as:

$$
\begin{aligned}
L^\intercal C L &= I && && (3.1) \\
C L &= {L^\intercal}^{-1} && \triangleright \text{left multiply by } {L^\intercal}^{-1} && (3.2) \\
C &= {L^\intercal}^{-1} L^{-1} && \triangleright \text{right multiply by } L^{-1} && (3.3) \\
C &= (LL^\intercal)^{-1} && \triangleright \text{since } \forall A, B (AB)^{-1} = B^{-1} A^{-1} \text{ and } {A^\intercal}^{-1} = {A^{-1}}^\intercal && (3.4) \\
C^{-1} &= LL^\intercal. && && (3.5)
\end{aligned}
$$

What we get then is that the linear transformation L that we are looking for is such that when multiplied by its own transpose gives us the inverse of the covariance matrix of our dataset, i.e., C^{-1}. Ma-

MATH REVIEW | SCATTER AND COVARIANCE MATRIX

Scatter matrices and **covariance matrices** are related concepts for providing descriptive statistics of datasets. Both matrices quantify the extent to which pairs of random variables from a multivariate dataset deviate from their respective means.
The only difference between the two is that the scatter matrix quantifies the above information in a cumulative manner, whereas in the case of covariance matrix, an averaged quantity normalized by the number of observations in the dataset is reported. By definition the scatter matrix of a data matrix X is given by

$$S = \sum_{i=1}^{n} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^\intercal,$$

with $\mathbf{x}_i$ and $\boldsymbol{\mu}$ denoting the i^{th} multivariate data point and the mean data point, respectively. Similarly the covariance matrix is given as

$$C = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^\intercal = \frac{1}{n} S.$$

A matrix M is called symmetric if $M = M^\intercal$ holds, i.e., the matrix equals its own transpose. Symmetry of both S and C trivially follows from their respective definitions and the fact that for any matrix $(AB)^\intercal = B^\intercal A^\intercal$.
A matrix M is a positive (semi)definite one whenever the inequality

$$\mathbf{y}^\intercal M \mathbf{y} \geq 0$$

relation holds. This property naturally holds for any scatter matrix S, since $\mathbf{y}^\intercal S \mathbf{y}$ is nothing else but a sum of squared numbers as illustrated below.

$$\begin{aligned} \mathbf{y}^\intercal S \mathbf{y} =& \mathbf{y}^\intercal \Big(\sum_{i=1}^{n} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^\intercal \Big) \mathbf{y} = \\ & \sum_{i=1}^{n} \mathbf{y}^\intercal (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^\intercal \mathbf{y} = \\ & \sum_{i=1}^{n} (\mathbf{y}^\intercal (\mathbf{x}_i - \boldsymbol{\mu}))^2 \geq 0 \end{aligned}$$

As the definition of the covariance matrix only differs in a scalar multiplicative factor, it similarly follows that expressions of the form $\mathbf{y}^\intercal C \mathbf{y}$, involving an arbitrary vector $\mathbf{y}$ and covariance matrix C can never potentially become negative.

Figure 3.5: Scatter and covariance matrix

trix L can be obtained by relying on the so-called Cholesky decomposition.

MATH REVIEW | CHOLESKY DECOMPOSITION

In linear algebra, **Cholesky decomposition** is a matrix factorization method, which can be applied for **symmetric**, **positive (semi)definite** matrices. We say that a matrix M is symmetric, if it equals to its own transpose, i.e., $M = M^\intercal$. A matrix is called positive (semi)definite, if $\mathbf{y}^\intercal M\mathbf{y} \geq 0$ (for every $\mathbf{y} \neq \mathbf{0}$).
If the above two conditions hold, then M can be decomposed in a special way into the product of two triangular matrices, i.e.,

$$M = LL^\intercal,$$

where L denotes some lower triangular matrix and $L^\intercal$ is the transpose of L (hence an upper triangular matrix). A matrix is said to be lower (upper) triangular if it contains non-zero elements only in its main diagonal and below (above) it and the rest of its entries are all zeros.

Figure 3.6: Cholesky decomposition

Example 3.2. *Determine the Cholesky decomposition of the matrix*

$$M = \begin{bmatrix} 4 & 2 \\ 2 & 1.25 \end{bmatrix}.$$

We know that the matrix we decompose M into has to be a lower and upper triangular matrix such that they are the transpose of each other. That explicitly being written out means that

$$M = \begin{bmatrix} 4 & 2 \\ 2 & 1.25 \end{bmatrix} = \begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} \\ 0 & l_{22} \end{bmatrix} = \begin{bmatrix} l_{11}^2 & l_{11} \cdot l_{21} \\ l_{21} \cdot l_{11} & l_{21}^2 + l_{22}^2 \end{bmatrix}.$$

From this, we immediately see that the value for l_{11} has to be chosen as $\sqrt{m_{11}} = \sqrt{4} = 2$. This means that we are one step closer to our desired decomposition. By substituting the value we determined for l_{11}, we get

$$M = \begin{bmatrix} 4 & 2 \\ 2 & 1.25 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} 2 & l_{21} \\ 0 & l_{22} \end{bmatrix} = \begin{bmatrix} 4 & 2 \cdot l_{21} \\ 2 \cdot l_{21} & l_{21}^2 + l_{22}^2 \end{bmatrix},$$

from where we can conclude that $l_{21} = \frac{m_{12}}{l_{11}} = \frac{2}{2} = 1$ is the proper choice. This time we got one further step closer to find the correct values for the lower and upper triangular matrices that we are looking for. If we substitute now the value determined for l_{21} we now have

$$M = \begin{bmatrix} 4 & 2 \\ 2 & 1.25 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & l_{22} \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 0 & l_{22} \end{bmatrix} = \begin{bmatrix} 4 & 2 \cdot 1 \\ 2 \cdot 1 & 1 + l_{22}^2 \end{bmatrix}.$$

We can now conclude that $l_{22} = \sqrt{1.25 - 1} = \sqrt{0.25} = 0.5$*, hence we managed to decompose the original matrix* M *into the product of a lower and upper triangular matrices which are being transposes of each other in the following form:*

$$M = \begin{bmatrix} 4 & 2 \\ 2 & 1.25 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 0.5 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 0 & 0.5 \end{bmatrix}.$$

3.3 Information theory and its application in data mining

We next review important concepts from information theory and their potential utilization when dealing with datasets.

3.3.1 Mutual information

Mutual information between two random variables X and Y is formally given as

$$MI(X;Y) = (H(X) + H(Y)) - H(X,Y), \tag{3.6}$$

i.e., it is the difference between the sum of the Shannon entropy for the individual variables, $H(X)$ and $H(Y)$, and their joint entropy, $H(X,Y)$.

After manipulating the formula in Eq. (3.6) a bit, we get that

$$MI(X;Y) = \sum_{x \in X} \sum_{y \in Y} P(X = x, Y = y) \log \frac{P(X = x, Y = y)}{P(X = x)P(Y = y)}. \tag{3.8}$$

What mutual information tells us about a pair of random variables X and Y is the amount of uncertainty which can be eliminated once the true value of either of the random variables is revealed to us. In other words, knowing about the outcome of a random variable, this is the amount of uncertainty which remains regarding the remaining random variable.

Example 3.3. *Imagine there are two random variables describing the weather conditions (W) and the mood of our boss (M). Throughout a series of 100 days, we jointly keep track of the daily weather and the mood of our boss and we record the* **contingency table**[10] *given in Table 3.5.*

[10] Contingency tables store the observation statistics of multiple random variables.

Table 3.5: An example contingency table for two variables.

	M=happy	M=blue	**Total**
W=sunny	38	7	**45**
W=rainy	12	43	**55**
Total	**50**	**50**	**100**

As illustrated by Table 3.5, without knowing anything about the weather conditions, we are totally clueless about the mood of our boss on average

MATH REVIEW | SHANNON ENTROPY

Shannon entropy is a quantity which tells us about the *expected* unpredictability of some random variable X. When X is a discrete random variable, it is formally defined as

$$H(X) = \sum_{x \in X} P(X = x) \log_2 \frac{1}{P(X = x)}. \tag{3.7}$$

The logarithmic part in Eq. (3.7), as illustrated in Figure 3.8, can be thought as a quantity which measures how surprised we get when we observe an event with probability $P(X = x)$. Indeed, if we assume that observing a certain event has probability 1.0, we should not get surprised at all since $\log_2 \frac{1}{1} = 0$. On the other hand, if we observe something with an infinitesimally small value ϵ, we should definitely become very surprised by the observation of such an event, which is also reflected by the quantity $\lim_{\epsilon \to 0} \log_2 \frac{1}{\epsilon} = \infty$.
Summing this quantity over the possible range of X and weighting each term with the probability of observing that value can be interpreted as an expected amount of surprise.
When we have more than just a single random variable, say X and Y, the concept of entropy can naturally be expanded to get their so-called joint entropy, i.e.,

$$H(X, Y) = \sum_{x \in X} \sum_{y \in Y} P(X = x, Y = y) \log_2 \frac{1}{P(X = x, Y = y)}.$$

Figure 3.7: Shannon entropy

since the marginal distribution of the mood random variable behaves totally unpredictably, i.e.,

$$P(M = happy) = P(M = blue) = 0.5.$$

This unpredictably is also reflected by the fact that the entropy of the random variable M takes its maximal possible value, i.e.,

$$H(M) = -0.5 \log_2 0.5 - 0.5 \log_2 0.5 = 1.$$

Analogous calculations result in $H(W) = 0.993$, $H(M, W) = 1.690$ and $MI(M, W) = 0.303$.

3.3.2 *Applying mutual information to data mining: Feature selection*

Imagine that we are performing **classification**, that is, we are given a series of multidimensional vectors that describe objects based on their predictive features, based on which, our goal is to infer some special categorical target variable about the objects. For instance,

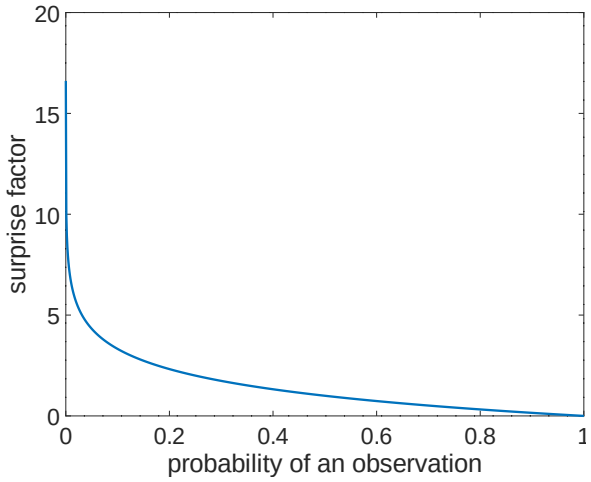


Figure 3.8: The amount of surprise for some event as a function of the probability of the event.

we might be given information about the customers of a financial institution who apply for a loan, and we want to decide in advance for an upcoming applicant who is described by the feature vector $\mathbf{x_i}$ whether this customer is a safe choice to provide the loan for. In that case, the categorical variable that we are about to predict would be a binary one, indicating whether the applicant is likely going to be able to repay the load (class label *Yes*) or not (class label *No*).

While solving a classification problem, we might want to reduce the number of predictive features or simply order them according to their perceived utility towards predicting the target class variable. The goal of **feature selection** is to choose the best subset of the predictors/features for our data mining application. Calculating mutual information is one of the many options to quantify the usefulness of predictive features towards a target variable (often denoted by Y in the literature).

Molina et al. [2002][11] provides a thorough survey of the alternative approaches for finding the best performing subset of the predictive features for a given task. Note that the task of feature selection is a hard problem, since when we have m features, there are exponentially many (2^m) possibilities to formulate subsets of features, for which reason heuristics to speed up the process are employed most of the times.

[11] Luis Carlos Molina, Lluís Belanche, and Àngela Nebot. Feature selection algorithms: A survey and experimental evaluation. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, pages 306–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1754-4. URL http://dl.acm.org/citation.cfm?id=844380.844722

3.3.3 *Applying mutual information to data mining: Feature discretization*

When representing our data with continuous features, it is sometimes also desired to turn the continuous features into discrete ones. This process is called **feature discretization**, meaning that instead of measuring the actual numeric values for a particular random variable, we transform its range into discrete bins and create a feature value

which indicates which particular interval of values a particular observation falls into. That is, instead of treating the salary of a person as a specific numeric value, we can form three bins of the salaries observed in our dataset (low, medium and high) and represent the salaries of the individuals by the range it falls.

The question is now, how to determine the intervals which form the discrete bins for a random variable? There are multiple answers to this question. Some of the approaches are uninformed (also called unsupervised) in the sense that the bins we split the range of our random variable is formed without considering that the different feature values might describe such data points that belong to a different target class variable $y \in Y$. These simple forms of feature discretization might strive for determining such bins of feature ranges that an equal amount of observations belong into each bin. A different form of partitioning can go along the formulation of bins with equal widths. These forms of equipartitioning approaches all have their potential drawbacks that information theory–based approaches can remedy.

A more principled way for feature discretization relies on mutual information. We can quantify with the help of mutual information the effects of performing discretization over X when assuming different values as boundaries for our bins. By calculating the various mutual information scores that we get if we perform the discretization fo X at various thresholds, we can select the boundary that is the most advantageous.

The mutual information–based feature discretization operates by calculating mutual information between a categorical class label Y and the discretized versions of X that we obtain by binning the observation into discrete categories at different thresholds. The choice for the threshold providing us with the highest mutual information can be regarded as the most meaningful way to form the different discrete intervals of our initially numeric variable X. Notice that this mutual information–based approach is more informed compared to the simple unsupervised equipartitioning approach, since it also relies on the class labels of the observations Y, not only the distribution of X. For this reason the mutual information–based discretization belongs to the family of informed (or supervised) discretization techniques.

Example 3.4. *Imagine that we have some numeric feature that we have 10 measurements from originating from 10 distinct instances. The actual numeric values observed for the 10 data points are depicted in Figure 3.9. Besides the actual values feature X takes on for the different observations, Figure 3.9 also reveals their class label Y. This information is encoded by the color of the dots representing each observation.*

Let us compare the cases when we form the two bins of the feature X to be

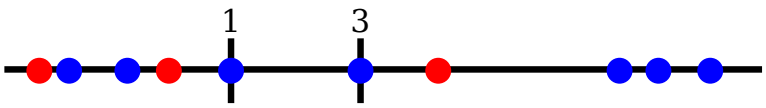


Figure 3.9: Sample feature distribution with observations belonging into two possible classes (indicated by blue and red).

$(-\infty, x]$ *and* (x, ∞)*, with* x *either being* 1 *or* 3*. As a first step, we need to calculate the contingency tables for the two potential locations of spitting* X*. The contingency tables for the two cases are included in Table 3.6, and they inform us about the number of feature values that fall into a given range of* X *broken down for the different class labels.*

	Class label Y	
	Red	Blue
$X \leq 1$	2	3
$X > 1$	1	4

(a) Using threshold $X \leq 1$

	Class label Y	
	Red	Blue
$X \leq 3$	2	4
$X > 3$	1	3

(b) Using threshold $X \leq 3$

Table 3.6: Sample dataset for illustrating feature discretization using mutual information.

We can now calculate the different mutual information scores corresponding to the cases of discretizing X *using the criteria* $X \leq 1$ *or* $X \leq 3$*. By denoting the discretized random variable that we derive from* X *by using a threshold for creating the bins to be* $t \in \{1, 3\}$ *by* X_t*, we get that*

$$MI(X_1, Y) = \frac{2}{10}\log_2\frac{4}{3} + \frac{3}{10}\log_2\frac{6}{7} + \frac{1}{10}\log_2\frac{2}{3} + \frac{4}{10}\log_2\frac{8}{7} \approx 0.0349$$

and

$$MI(X_3, Y) = \frac{2}{10}\log_2\frac{10}{9} + \frac{4}{10}\log_2\frac{20}{21} + \frac{1}{10}\log_2\frac{5}{6} + \frac{3}{10}\log_2\frac{15}{14} \approx 0.0058.$$

We arrived to the above results by applying Eq. (3.8) for the data derived from the contingency tables in Table 3.6. Based on the values obtained for $MI(X_1, Y)$ *and* $MI(X_3, Y)$*, we can conclude that – according to our sample – performing discretization using the threshold* $t = 1$ *is a better choice.*

? Can you find another data point from Figure 3.9, using which as a boundary for discretization performs even better than the choice of $t = 1$?

3.4 Eigenvectors and eigenvalues

The concepts of **eigenvectors** and **eigenvalues** frequently reoccur during our discussion of various topics later on related to e.g. dimensionality reduction (Chapter 6) and graph-based data mining approaches (Chapter 8). In order to ease the understanding of those parts, we revisit here these important concepts next briefly.

Given a square matrix $M \in \mathbb{R}^{n \times n}$, an eigenvalue–eigenvector pair for M satisfies the following equality

$$M\mathbf{x} = \lambda\mathbf{x}, \tag{3.9}$$

for some scalar λ and n-dimensional vector $\mathbf{x}$. Note that any $n \times n$ matrix has n (not necessarily distinct) eigenvalue–eigenvector pairs. To this end, we shall index the different eigenvalues and eigenvectors a matrix has as $\lambda_i, \mathbf{x}_i$ ($1 \leq i \leq n$). In the general case, the λ_i values can be complex as well, however, matrices that we consider in this book can always be assumed to have real eigenvalues.

Intuitively, an eigenvector of matrix M is such a vector, the direction of which does not get altered – modulo to reflection perhaps – relative to its original orientation. Although the direction of the eigenvectors remains intact, their magnitude can change. The rate with which an eigenvector changes is exactly its corresponding eigenvalue. This means that if an eigenvector $\mathbf{x}$ has a corresponding eigenvalue of 2, then all the components of the matrix–vector product $M\mathbf{x}$ would be twice the original coordinates of $\mathbf{x}$.

It turns out that the eigenvalues for matrix M are such values λ, which satisfy that $\det(M - \lambda I) = 0$, where I denotes the identity matrix and det refers to the determinant of its argument.

We can see, that the definition of an eigenvalue according to Eq. (3.9) implies that the equation

$$(M - \lambda I)\mathbf{x} = \mathbf{0} \tag{3.10}$$

also has to hold, with $\mathbf{0}$ marking the vector of all zeros. This kind of homogeneous system of linear equation can be trivially solved by $\mathbf{x} = \mathbf{0}$. This solution is nonetheless a trivial one, which works for any M. If we wish to avoid obtaining such a trivial – hence uninteresting – solution, the rows of $(M - \lambda I)$ should not be linearly independent. Had $(M - \lambda I)$ be of full rank (meaning that it consisted of linearly independent rows), the only solution which would satisfy Eq. (3.10) would be the trivial one. The way to ensure $(M - \lambda I)$ not to be of full rank, is to require $\det(M - \lambda I) = 0$ to hold. Determinants can be viewed as polynomials, hence the eigenvalues of M are the roots of the polynomial that we obtain from the determinant of the matrix $M - \lambda I$. The polynomial that we can construct from the determinant of matrix $(M - \lambda I)$ is called tha **characteristic polynomial** of M. Essentially, the λ eigenvalues for M are the roots of the characteristic polynomial derived from M.

Example 3.5. *As a concrete example, let us determine the eigenpairs of the matrix* $M = \begin{bmatrix} 5 & 1 \\ 4 & 8 \end{bmatrix}$. *Based on the previous discussion, the eigenvalues of* M *need to be such that the determinant of the matrix*

$$M - \lambda I = \begin{bmatrix} 5 - \lambda & 1 \\ 4 & 8 - \lambda \end{bmatrix}$$

equals zero. Calculating the determinant of a $2x2$ *matrix is easy, as all we need to do is to multiply its elements across its diagonal and subtract the product of the elements in the off-diagonal, leaving us with the polynomial*

$$p(\lambda) = (5-\lambda)(8-\lambda) - 1*4 = \lambda^2 - 13\lambda + 36.$$

Finding the roots of the above quadratic equation, we get that $\lambda_1 = 4$ *and* $\lambda_2 = 9$*. If we substitute back, we get that the eigenvector* $\mathbf{x}$ *accompanying the eigenvalue* $\lambda = 4$ *has to fulfill*

$$\begin{bmatrix} 1 & 1 \\ 4 & 4 \end{bmatrix} \mathbf{x} = \mathbf{0},$$

which is a system of two linear equations and two unknowns.

Since we deliberately constructed our system of equation such that it consist of linearly dependent row coefficients, we are free to choose one of the variables in $\mathbf{x}$ *to any value. Let us hence arbitrarily set* x_2 *to 1.*

What it means – because of the $x_1 + x_2 = 0$ *and* $4x_1 + 4x_2 = 0$ *requirements in our system of equations – is that* $x_1 = -x_2$*, meaning that the eigenvector corresponding the eigenvalue* 4 *is* $\mathbf{x} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$.

Note that not only vector $\mathbf{x}$ *but any other* $c\mathbf{x}$ *with* $c \in \mathbb{R}$*, say* $\begin{bmatrix} 2.2 \\ -2.2 \end{bmatrix}$*, would yield a valid solution to the above linear system of equations. In order to avoid this ambiguity, a common practice is to think of and report eigenvectors such that they have a unit norm. What it means, that the canonical way of reporting the eigenvector that we just found is to divide all of its components by its norm, i.e.,* $\sqrt{2}$ *in the given example. What it means is that one of the eigenvalue–eigenvector pairs for matrix* M *is*

$$\left(4, \begin{bmatrix} \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}\right).$$

After a similar line of thought, we get for the other eigenvalue that

$$\begin{bmatrix} -4 & 1 \\ 4 & -1 \end{bmatrix} \mathbf{x} = \mathbf{0}$$

also has to hold. By choosing x_2 *to be 1, we get that* x_1 *has to be* $\frac{1}{4}$*. This means that the eigenvector corresponding to the eigenvalue* 9 *for matrix* M *is* $\begin{bmatrix} \frac{1}{4} \\ 1 \end{bmatrix}$*, or in its unit normalized canonical form* $\begin{bmatrix} \frac{1}{\sqrt{17}} \\ \frac{4}{\sqrt{17}} \end{bmatrix} \approx \begin{bmatrix} 0.2425 \\ 0.9701 \end{bmatrix}$.

Let us now also see how can we obtain the eigenpairs of matrix M using Octave. Figure 3.10 and Figure 3.11 provides two alternative ways for doing so.

Figure 3.10 illustrates the usage of the convenient built-in function of Octave for calculating eigenproblems. The default behavior of the function `eig` is that it returns the eigenvalues of its argument in the form of a vector. In case we also want to know the eigenvectors corresponding to the particular eigenvalues, we can also do so, by using not only the default return value of the function, but the tuple of matrices it can also return. In the latter case the matrix returned second contains the eigenvalues of the argument in its main diagonal, and the matrix returned at position one contains one (unit-normalized) eigenvector in each of its columns. The eigenvalue in the same position in the second returned matrix corresponds to the eigenvalue in the same column from the first returned matrix.

CODE SNIPPET

```
M=[5 1; 4 8];
# obtaining only the eigenvalues of M
eigenvals = eig(M)
>> eigenvals =
  4
  9

# obtaining both the eigenvectors and eigenvalues of M
[eigenvecs, eigenvals] = eig(M)
>> eigenvecs =

  -0.70711  -0.24254
   0.70711  -0.97014

eigenvals =

Diagonal Matrix

   4   0
   0   9
```

Figure 3.10: Eigencalculation using Octave

We provide an alternative way for calculating the eigenpairs of the same matrix M in Figure 3.11. Notice how the provided calculation connects to the calculation provided in Example 3.5. In Figure 3.11, the function `roots` is used for finding the roots of a polynomial determined by its coefficients of decreasing degree and the function `null` finds (unit-normalized) vectors in the **null space** of its argument. Recall that a null space of some matrix A are such vectors $\mathbf{x}$ for which $A\mathbf{x} = \mathbf{0}$ holds.

? Can you anticipate the values for `eig_vals`, `eig_vec1` and `eig_vec2` in Figure 3.11? Hint: looking back at Example 3.5 and Figure 3.10 can help a lot in giving the right answer (potentially modulo to a multiplier -1 for `eig_vec1` and `eig_vec2`)?

CODE SNIPPET

```
eig_vals=roots([1 -13 36]);
eig_vec1=null(M-eig_vals(1)*eye(size(M)));
eig_vec2=null(M-eig_vals(2)*eye(size(M)));
```

Figure 3.11: An alternative way of determining the eigenpairs of matrix M without relying on the built-in Octave function `eig`.

3.5 *Summary of the chapter*

This chapter introduced the basic concepts and fundamental techniques for data representation and transformation in data mining. At the end of the chapter, we revisited the problem of eigencalculation for matrices, a technique that we will refer to multiple times throughout the remainder of the book.

4 | Distances and similarities

Learning Objectives:
- Distance metrics
- Mahalanobis distance
- Cosine distance
- Minkowski distance
- Jaccard distance
- Edit distance
- Distances for distributions

Readers of this chapter can learn about different ways of quantifying similarity between pairs of objects. In particular, by the end of the chapter one should

- recall and explain various similarity measures,
- justify their choice towards a specific similarity measure for particular kinds of datasets.

Determining (dis)similarity between pairs of objects have clear application possibilities. Imagine you own a web shop and a certain user buys some product X. We can easily recommend our users further products they might be interested in purchasing by simply having a good notion of similarity between the objects. As another possible use case where having a good notion of similarity can be helpful, is looking for plagiarism. In that case, having a high degree of similarity between two essays submitted for evaluation might be a good sign for breaching the code of conduct by the authors of such essays.

Intuitively, and thinking in geometric terms, we can say that the closer two points are located to each other, the more similar they are. As we start thinking about closeness, it is natural to introduce **distance metrics**. Formally, a function $d : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}$ defined over the k-dimensional point pair (a, b) is a distance metric if the following axioms hold:

1. $d(a, b) \geq 0$ (non-negativity)
2. $a = b \Leftrightarrow d(a, b) = 0$ (positive definiteness)
3. $d(a, b) = d(b, a)$ (symmetry)
4. $d(a, b) \leq d(a, c) + d(c, b)$ for *any* point c (triangle inequality).

Example 4.1. *Let us verify that the distance defined as the cardinality of symmetric set difference fulfills the axioms of distance metrics. As a*

reminder, the symmetric difference between two sets A and B is defined as $A \triangle B = (A \setminus B) \cup (B \setminus A)$. Supposing $A = \{a, b, c\}$ and $B = \{b, c, d, e\}$, the symmetric difference of the two sets is then the set $A \triangle B = \{a, d, e\}$, i.e., the set of those elements that are present in exactly one of the input arguments.

1. *The non-negativity trivially holds as the result of symmetric set difference is another (possibly empty) set, the cardinality of which is always non-negative.*

2/a $\Rightarrow$ *When set $A = B$ holds, both $A \setminus B = B \setminus A = \emptyset$, hence the distance equals the cardinality of the empty set, i.e., 0. Note that the second property is not just an implication, but an equivalence for which reason the opposite direction has to be verified as well.*

2/b $\Leftarrow$ *Given that the distance is 0, we need to see that $A = B$ has to hold. In order to see that, notice that*

$$\begin{aligned} |A \triangle B| &= |A \setminus B| + |B \setminus A| && \text{by definition of } \triangle \\ &= (|A| - |A \cap B|) + (|B| - |A \cap B|) && \text{by definition of } \setminus \\ &= (|A \cup B| + |A \cap B|) - 2|A \cap B| && \text{from the Inclusion-Exclusion Principle} \\ &= |A \cup B| - |A \cap B|. \end{aligned}$$

This means that whenever $|A \triangle B| = 0$, $|A \cup B| = |A \cap B|$ also has to hold, which is only true when $A = B$.

3 *The symmetry follows from the definition of the distance and the commutativity of addition, i.e.,*

$$|A \triangle B| = |A \setminus B| + |B \setminus A| = |B \setminus A| + |A \setminus B| = |B \triangle A|.$$

4 *Finally, we have to show that the triangle inequality holds for any sets A, B and C.*

$$\begin{aligned} |A \triangle B| + |B \triangle C| = \\ (|A| + |B| - 2 \cdot |A \cap B|) + (|B| + |C| - 2 \cdot |B \cap C|) = \\ |A| + |C| + 2 \cdot |B| - 2 \cdot |A \cap B| - 2 \cdot |B \cap C| \geq \\ |A| + |C| - 2 \cdot |B \cap C| \geq \\ |A| + |C| \geq \\ |A| + |C| - 2 \cdot |A \cap C| = |A \triangle C|. \end{aligned}$$

For those who like visual arguments, Figure 4.1 can also provide evidence for the triangle inequality being fulfilled. The intuitive explanation here is that in the Venn diagram, there is no are which would be covered by green stripes (corresponding to $A \triangle C$) and not covered by either of blue stripes (corresponding to $A \triangle B$) or red stripes (corresponding to $B \triangle C$).

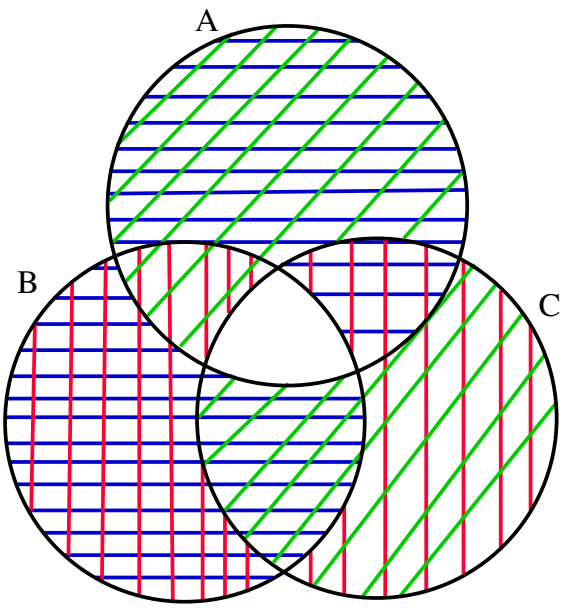


Figure 4.1: Illustration for the symmetric set difference fulfilling the triangle inequality.

We now list a few of the most important distances. We should stress, however, that there is a wide variety of further distances that we cannot cover here due to space constraints. Readers with a keen interest in various additional distances can find a comprehensive list to read in [1].

[1] Deza and Deza 2009

4.1 *Minkowski distance*

Minkowski distance of order p between two data points $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^k$ is formally defined as

$$d_p(\boldsymbol{x}, \boldsymbol{y}) = \Big(\sum_{i=1}^{k} |\boldsymbol{x}_i - \boldsymbol{y}_i|^p \Big)^{\frac{1}{p}} \tag{4.1}$$

The Minkowski distance of order p between two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is basically the p-norm of their difference $\boldsymbol{x} - \boldsymbol{y}$. It is instructive to imagine unit circles for various p values as depicted in Figure 4.2 in order to gain a better intuition of different norms and the Minkowski distance of different orders.

It is worth noting that the norm for $p < 1$ is non-convex as also illustrated by Figure 4.2. It is also easy to see that when $p < 1$, the triangle inequality is not met.

Example 4.2. *Imagine the following three points* $\boldsymbol{a} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $\boldsymbol{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ *and*

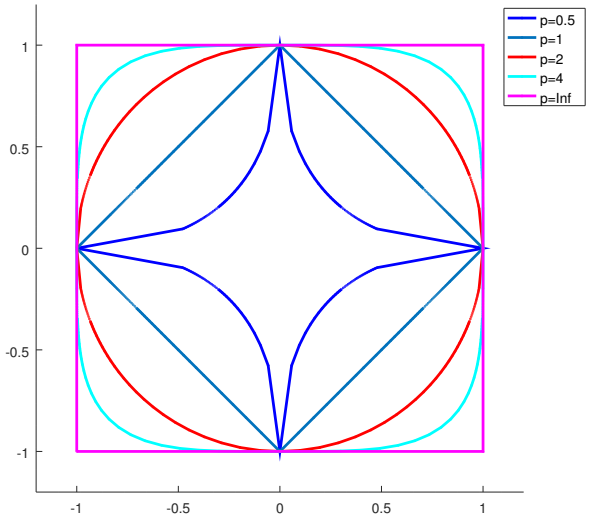


Figure 4.2: Unit circles for various p norms.

$\boldsymbol{c} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ *and let the order* p *be* 0.1. *As for the pairwise Minkowski distances of order* 0.1 *we then get*

$$d_{p=0.1}(\boldsymbol{a}, \boldsymbol{b}) = (1^{0.1} + 1^{0.1})^{10} = 2^{10} = 1024$$
$$d_{p=0.1}(\boldsymbol{a}, \boldsymbol{c}) = (1^{0.1} + 0^{0.1})^{10} = 1^{10} = 1$$
$$d_{p=0.1}(\boldsymbol{b}, \boldsymbol{c}) = (0^{0.1} + 1^{0.1})^{10} = 1^{10} = 1,$$

which values do not conform with the triangle inequality as the distance from a to b is larger than the sum of distances between a and c and c to b. What this (un)intuitively suggests is that you can find shorter routes between two points than simply directly going from the source point to the target destination by stopping at some – appropriately chosen – intermediate location.

Minkowski distance is most frequently applied for order $p \geq 1$, primarily because convexity and triangle equality is assured in those cases. Three particular choices for p are so typical that for them, there is also a special name, i.e.,

- for $p = 1$, we get the **Manhattan distance** (or the **city block distance**)
- for $p = 2$, we get the **Euclidean distance**
- for $p = \infty$, we get **Chebyshev distance**.

Example 4.3. *Given points* $\boldsymbol{a} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$ *and* $\boldsymbol{b} = \begin{bmatrix} 6 \\ 4 \end{bmatrix}$ *as depicted in Figure 4.3, calculate the Minkowski distance of order* $p \in \{1, 2, 2.5, \infty\}$.

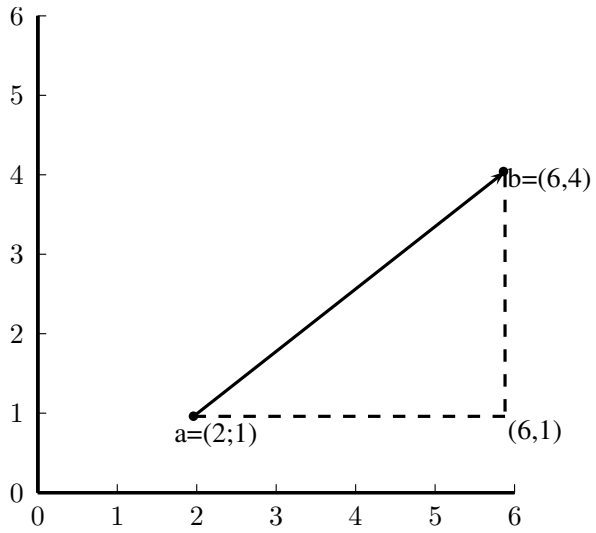


Figure 4.3: Example points for illustrating the Minkowski distance for different values of p.

$$
\begin{aligned}
d_{p=1}(\boldsymbol{a},\boldsymbol{b}) &= (4^1+3^1)^1 = 7\\
d_{p=2}(\boldsymbol{a},\boldsymbol{b}) &= (4^2+3^2)^{0.5} = \sqrt{25} = 5\\
d_{p=2.5}(\boldsymbol{a},\boldsymbol{b}) &= (4^{2.5}+3^{2.5})^{0.4} = 47.59^{0.4} \approx 4.69\\
d_{p=\infty}(\boldsymbol{a},\boldsymbol{b}) &= \max(4,3) = 4.
\end{aligned}
$$

Notice, that the $p = \infty$ case (i.e., the Chebyshev distance) boils down to the maximum of the absolute values of the dimension-wise coordinate differences between the two points. To understand intuitively, why this is the case when $p = \infty$, assume that m is the largest coordinate-wise difference in absolute value, that is

$$m = \max_{1 \leq i \leq k} |\boldsymbol{x}_i - \boldsymbol{y}_i|.$$

As p approaches infinity, we observe that the sum in the general definition of Minkowski distance is getting dominated by the term m^p. Since the contribution of all the other tems not related to m can be neglected as p goes to infinity, the entire expression in Eq. (4.1) boils down to $(m^p)^{\frac{1}{p}} = m$. Notice that even if the same amount of maximum absolute difference m happens to be the difference for $l > 1$ dimensions, we end up getting the same result, as in that case Eq. (4.1) can be expressed as

$$(lm^p)^{\frac{1}{p}} = l^{\frac{1}{p}}(m^p)^{\frac{1}{p}},$$

which is simply due to the fact that $\lim_{p\to\infty} l^{\frac{1}{p}} = \lim_{p\to\infty} \sqrt[p]{l} = 1$. Example 4.3 further reveals the monotonity of the Minkowski distance as the distances we obtain decrease in a monotone fashion as p increases.

4.2 *Mahalanobis distance*

The way Minkowski distance calculates distances is that it aggregates the dimension-wise discrepancies between data points, while assuming that the individual dimensions are totally independent of each other. The different dimensions, however, are most often correlated with each other to a varying degree. Due to the correlation being present between variables, Minkowski distance has the potential of "overcounting" dissimilarity between data points in some sense. Simultaneously, even a small difference along a dimension can be viewed as a large deviation for a certain pair of observations, if that particular deviation happens for such a dimension, where data points do not typically tend to differ in our dataset.

Figure 4.4 contains a small example dataset for which relying on such a distance function which motivates the incorporation of the correlation between the different features into the quantification of the distance between pairs of data points. Although point B from Figure 4.4 is located closer to point A as opposed to point C in the Minkowski notation of distances, it still sounds reasonable to argue for point B being more dissimilar to point A than point C.

Mahalanobis distance provides a theoretically motivated way for calculating distance in a way which guarantees that the dissimilarities that we sum up along the different dimensions are indeed independent of each other. This is achieved by incorporating a transformation which performs the **whitening** of our dataset (cf. Section 3.2.3) into the calculation of pairwise distances.

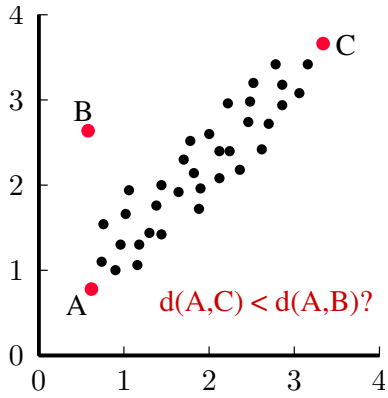


Figure 4.4: Motivation for the Mahalanobis distance.

As we have seen it earlier in Eq. (3.5) Section 3.2.3, we can transform multidimensional observations by such a matrix L, where L is the lower-triangular matrix originating from the Cholesky decomposition of the inverse of the covariance matrix of our dataset. Mahalanobis distance is defined between two vectors according to the

formula

$$d_M(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{(\boldsymbol{x} - \boldsymbol{y})^\intercal \Sigma^{-1} (\boldsymbol{x} - \boldsymbol{y})}, \tag{4.2}$$

where Σ^{-1} denotes the inverse of the covariance matrix of the data from which observations $\boldsymbol{x}$ and $\boldsymbol{y}$ come from.

Inspecting the formula of Mahalanobis distance, we can notice that it resembles Minkowski distance of order $p = 2$, i.e., standard Euclidean distance. Indeed, if the covariance matrix of our dataset happens to be the **identity matrix** I – a matrix of all ones in its diagonal and zeros otherwise –, then the formula in Eq. (4.2) exactly simplifies down to the Euclidean norm. This is because the Σ^{-1} term cancels out from Eq. (4.2) in the case when $\Sigma = I$.

? Based on the formula in Eq. 4.2, can you tell in what way does the Mahalanobis distance relate to some well-known probability distribution?

When the covariance matrix of some dataset is the identity matrix, it means that there is no correlation between the different coordinates anyway and the variances across all the dimensions are identical. As the inverse of the identity matrix is itself, Eq. (4.2) boils down to

$$\sqrt{(\boldsymbol{x} - \boldsymbol{y})^\intercal (\boldsymbol{x} - \boldsymbol{y})} = \sqrt{\|\boldsymbol{x} - \boldsymbol{y}\|_2^2} = \sqrt{\sum_{i=1}^{k} (x_i - y_i)^2} = \|\boldsymbol{x} - \boldsymbol{y}\|_2,$$

which is exactly the Euclidean distance between $\boldsymbol{x}$ and $\boldsymbol{y}$.

Let us notice, how Eq. (4.2) can be equivalently expressed as

$$\|L^\intercal \boldsymbol{x} - L^\intercal \boldsymbol{y}\|_2 = \|L^\intercal (\boldsymbol{x} - \boldsymbol{y})\|_2 =$$
$$\sqrt{(\boldsymbol{x} - \boldsymbol{y})^\intercal L L^\intercal (\boldsymbol{x} - \boldsymbol{y})} = \sqrt{(\boldsymbol{x} - \boldsymbol{y})^\intercal \Sigma^{-1} (\boldsymbol{x} - \boldsymbol{y})},$$

implying that Mahalanobis distance is essentially a special form of the Euclidean distance, where the vectors of the original space are first transformed into such a representation by matrix L, which makes the individual variables independent of each other. This behavior of the Mahalanobis distance is illustrated in Figure 4.5.

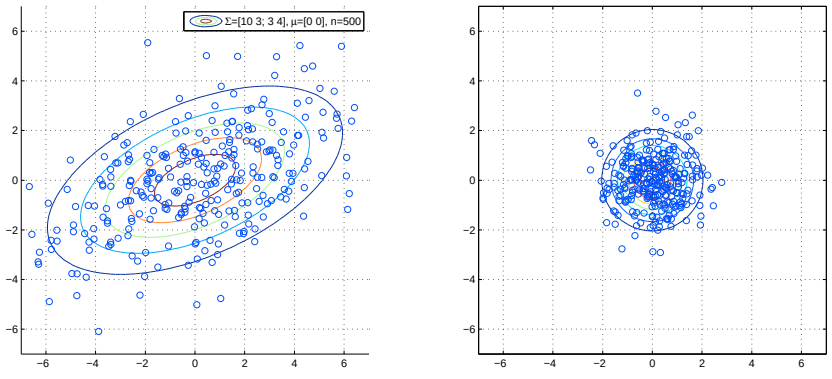


Figure 4.5: Illustration of the Mahalanobis distance.

? Can you think of some practical difficulties that can arise for calculating the Mahalanobis distance?

4.3 Cosine distance

The **cosine distance** is derived from the **cosine similarity**. Cosine similarity quantifies the cosine of the angle enclosed by a pair of vectors as illustrated in Figure 4.6 and can it is calculated based on the formula

$$cos(\theta) = \frac{\boldsymbol{x}^\intercal \boldsymbol{y}}{\|\boldsymbol{x}\| \|\boldsymbol{y}\|}. \tag{4.3}$$

We get cosine distance by taking the arccos function over the result of Eq. (4.3). Taking the arccos basically means that we are measuring the distance between two vectors as $\arccos(\cos(\theta))$, i.e., the angle enclosed by the two vectors. As a consequence, the larger the angle between a pair of vectors is, the larger their distance becomes. After all, the cosine distance is insensitive to the difference between the vectors in any other sense other than the rotational angle needed to get from the orientation of one to that of the other.

In Figure 4.6, we can see that the distance between the blue and red vectors is definitely non-zero in the Minkowski sense. The cosine distance between these two vectors is zero, however, as they point to the same direction, meaning that the angle enclosed by them is zero. This also suggest us that the cosine distance does not fulfill the $\boldsymbol{x} = \boldsymbol{y} \Leftrightarrow d(\boldsymbol{x}, \boldsymbol{y}) = 0$ property of metrics, however it obeys the milder property of $\boldsymbol{x} = \boldsymbol{y} \Rightarrow d(\boldsymbol{x}, \boldsymbol{y}) = 0$. Figure 4.6 further reveals that even though the Minkowski distance between the red and black vectors is smaller than that of the red and blue vectors, their cosine distances behave in the opposite manner. This is because cosine distance disregards the differences in the norm of the vectors and purely focuses on their orientation.

? Can you think of examples when measuring the dissimilarity of data points is better done by calculating their cosine distance instead of the Minkowski (e.g. Euclidean) distance?

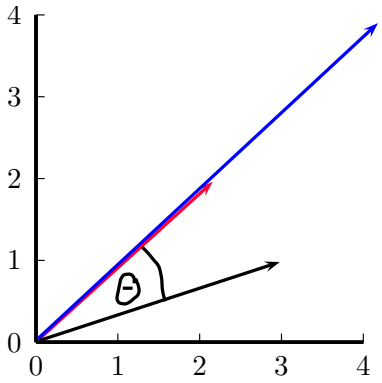


Figure 4.6: Illustration of the cosine distance

In order to see why Eq. (4.3) holds, one can rely on the Law of cosines summarized in Eq. (4.4). Thinking in terms of vector spaces, given two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, we get a triangle the three vertices are

MATH REVIEW | LAW OF COSINES

For any triangle with sides a, b and c, the Law of cosines says that the equality

$$c^2 = a^2 + b^2 - 2ab\cos(\theta) \tag{4.4}$$

holds, where θ denotes the angle enclosed by sides a and b. Disregarding the $2ab\cos(\theta)$ term, this equality is very likely to be reminiscent for most readers, since its omission leaves us with the well-known Pythagorean theorem. Indeed, when the angle enclosed by side a and b of the triangle is a right angle, the last term on the right side of the equation cancels out, exactly leaving us with the Pythagorean formula for right-angled triangles with hypotenuse c and legs a, b.

Figure 4.7: Law of cosines

the origin and the endpoints determining the two vectors. Now, the lengths of the three sides of this triangle are going to be $\|\boldsymbol{a}\|$, $\|\boldsymbol{b}\|$ and $\|\boldsymbol{a} - \boldsymbol{b}\|$. If we denote by θ the angle enclosed by $\boldsymbol{a}$ and $\boldsymbol{b}$ and apply the Law of cosines, what we get is that

$$\|\boldsymbol{a} - \boldsymbol{b}\|^2 = \|\boldsymbol{a}\|^2 + \|\boldsymbol{b}\|^2 - 2\|\boldsymbol{a}\|\|\boldsymbol{b}\|\cos(\theta). \tag{4.5}$$

Relying on the fact that $\|v\|^2 = v^\intercal v$ for any vector v, we can rewrite Eq. (4.5) as

$$(\boldsymbol{a} - \boldsymbol{b})^\intercal(\boldsymbol{a} - \boldsymbol{b}) = \boldsymbol{a}^\intercal\boldsymbol{a} + \boldsymbol{b}^\intercal\boldsymbol{b} - 2\|\boldsymbol{a}\|\|\boldsymbol{b}\|\cos\theta,$$

which equals

$$\boldsymbol{a}^\intercal\boldsymbol{a} + \boldsymbol{b}^\intercal\boldsymbol{b} - 2\boldsymbol{a}^\intercal\boldsymbol{b} = \boldsymbol{a}^\intercal\boldsymbol{a} + \boldsymbol{b}^\intercal\boldsymbol{b} - 2\|\boldsymbol{a}\|\|\boldsymbol{b}\|\cos\theta. \tag{4.6}$$

From that point, simple reordering of the two sides of Eq. (4.3) yields us Eq. (4.3).

Example 4.4. *Suppose we have three 4-dimensional points*

$$\boldsymbol{x_1} = \begin{bmatrix} \sqrt{2} \\ -1 \\ 2 \\ 3 \end{bmatrix}, \boldsymbol{x_2} = \begin{bmatrix} \sqrt{2} \\ 2 \\ -1 \\ \sqrt{2} \end{bmatrix}, \boldsymbol{x_3} = \begin{bmatrix} -\sqrt{2} \\ 1 \\ -2 \\ 3 \end{bmatrix}.$$

Let us calculate the pairwise cosine distances between these points. In order to do so, we first calculate the norms of the individual vectors:

$$\|\boldsymbol{x_1}\|_2 = \sqrt{\boldsymbol{x_1}^\intercal\boldsymbol{x_1}} = \sqrt{\sqrt{2}^2 + (-1)^2 + 2^2 + 3^2} = \sqrt{2+1+4+9} = \sqrt{16} = 4,$$

$$\|\boldsymbol{x_2}\|_2 = \sqrt{\boldsymbol{x_2^\intercal x_2}} = \sqrt{\sqrt{2}^2 + 2^2 + (-1)^2 + \sqrt{2}^2} = \sqrt{2+4+1+2} = \sqrt{9} = 3,$$

$$\|\boldsymbol{x_3}\|_2 = \sqrt{\boldsymbol{x_3^\intercal x_3}} = \sqrt{(-\sqrt{2})^2 + 1^2 + (-2)^2 + 3^2} = \sqrt{2+1+4+9} = \sqrt{16} = 4.$$

Next, we need to calculate the pairwise dot product between the vectors:

$$\begin{aligned}
\boldsymbol{x_1^\intercal x_2} &= \sqrt{2}\cdot\sqrt{2} + (-1)\cdot 2 + 2\cdot(-1) + 3\cdot\sqrt{2} \approx 2.243,\\
\boldsymbol{x_1^\intercal x_3} &= \sqrt{2}\cdot(-\sqrt{2}) + (-1)\cdot 1 + 2\cdot(-2) + 3\cdot 3 = 2,\\
\boldsymbol{x_2^\intercal x_3} &= \sqrt{2}\cdot(-\sqrt{2}) + 2\cdot 1 + (-1)\cdot(-2) + \sqrt{2}\cdot 3 \approx 6.243.
\end{aligned}$$

We can get the cosine distance for a pair of vectors if we take the arccos *of the fraction of their product and the product of their norms, i.e.,*

$$\begin{aligned}
d_{cos}(\boldsymbol{x_1}, \boldsymbol{x_2}) &= \arccos(2.243/12) \approx 79.23^\circ,\\
d_{cos}(\boldsymbol{x_1}, \boldsymbol{x_3}) &= \arccos(2/16) \approx 82.82^\circ,\\
d_{cos}(\boldsymbol{x_2}, \boldsymbol{x_3}) &= \arccos(6.243/12) \approx 58.65^\circ.
\end{aligned}$$

Based on our calculations so far, we can conclude now that vectors $\boldsymbol{x_2}$ and $\boldsymbol{x_3}$ are the closest ones to each other.

4.4 Jaccard similarity

Jaccard similarity (also referred as Jaccard coefficient) is a similarity score that is suited for measuring the overlap between objects described as sets. Given two sets X and Y, it is defined as

$$sim_{Jaccard}(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}. \tag{4.7}$$

As such, the Jaccard coefficient quantifies the relative cardinality of intersection between a pair of sets by normalizing with the cardinality of their union. The fraction gets maximized when the two sets contain the exact same elements, in which case the nominator and the denominator become the same, resulting in a similarity score of one. Unless $X = Y$ holds, $|X \cap Y| < |X \cup Y|$ surely needs to hold, for which reason the Jaccard similarity never exceeds 1.0. As an example, the two sets illustrated by their Venn diagrams in Figure 4.8 has a Jaccard similarity of $\frac{2}{10}$.

The smallest possible value the Jaccard similarity can take between two points is obviously zero: this happens when there is absolutely no overlap between the two sets it is calculated for. This means that

$$0 \leq sim_{Jaccard}(X, Y) \leq 1,$$

for every set X and Y. It reaches the value 1 only if the two sets are the same and has value 0 if the two set has no elements in common.

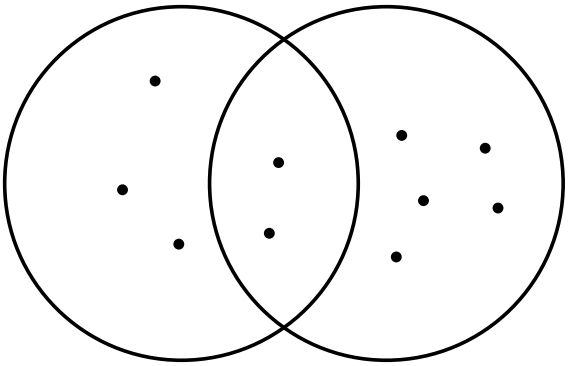


Figure 4.8: An example pair of sets X and Y.

Jaccard similarity can be extended to multisets as well, i.e., to such cases, when the set is allowed to store its members multiple times. In such a case the formula for this generalized Jaccard similarity becomes $sim_{Jaccard}(A,B) = \frac{\sum_i \min(X_i,Y_i)}{\sum_i \max(X_i,Y_i)}$, where X_i and Y_i denotes the presence of item i in set X and Y, respectively.

The **Dice similarity** is a very close variant of the Jaccard coefficient, which is calculated according to the formula

$$sim_{Dice}(X,Y) = \frac{2|X \cap Y|}{|X| + |Y|}. \tag{4.8}$$

There is a close relation between the Dice and the Jaccard coefficients according to

$$sim_{Dice}(X,Y) = \frac{2 \cdot sim_{Jaccard}(X,Y)}{1 + sim_{Jaccard}(X,Y)}, \tag{4.9}$$

meaning that once someone calculates the Jaccard coefficient between two sets, it is needless to do the calculation for the Dice coefficient as well, since by substituting into Eq. (4.9), one can directly get the answer.

Since the two measures are so closely related to each other, it suffices to deal with one of them in the followings. The similarity that we are going to analyze from closer is going to be Jaccard similarity.

Example 4.5. *It is quite typical to represent textual documents as sets, i.e., based on the words or character* **n-gram***s that are found is some particular text. In this sense any text, e.g. an essay or a novel, can be viewed as a (multi)set of the n-grams (also called* **shingles***) that can be found in the text. This means for instance that choosing* $n = 3$*, the sentence 'I am happy.' is represented by the following set of character trigrams: {'I a', ' am', 'am ', 'm h', ' ha', 'hap', 'app', 'ppy', 'py.'}.*

Note that the choice for n when determining shingles can be decisive in accessing similarity between two sets describing documents. If

we choose n to be too low, then most documents will become highly similar to each other. In the most extreme case, when $n = 1$, documents would simply be described by the set of characters they contain. As such, it is very likely that every document would have a large set of intersecting characters. On the other hand, by selecting n to be a fairly large number, we would hardly find any overlapping shingles between document pairs.

The so-called **hashing trick** is often applied to shingles. What hashing trick does is that it partitions distinct shingles into equivalence classes and instead of modeling documents by their actual character n-grams, documents get represented by the equivalence classes of their character n-grams. Identifying shingles by their hash values can serve the purpose of saving us memory as we are no longer distinguishing every single character n-gram.

For deriving a distance from either of the similarities discussed in this section, we can simply subtract from the maximum possible similarity value, i.e., 1.0 their true similarity in order to get a distance.

4.5 *Edit distance*

Edit distance measures the dissimilarity between a pair of strings, i.e., given two character sequences S_1 and S_2, it tells us the number of insertion and deletion operations which has to be performed in order to turn string S_1 into S_2. For instance, given the character sequences $S_1 = lean$ and $S_2 = lap$, their edit distance is 3. To see why, notice the following derivation

$$lean \xrightarrow{\text{delete e}} lan \xrightarrow{\text{delete n}} la \xrightarrow{\text{insert p}} lap.$$

In some sense, it can be thought as a difference between multisets as well, i.e., we have to turn the characters in S_1 into those of S_2, however, we can save us the effort of rewriting certain characters, i.e., those which can be found in both strings in the same order. We can think of the order-preserving overlapping characters as the "intersection" between the two strings. More precisely, this intersection is called the **longest common subsequence** between the two strings. Having introduced the concept of longest common subsequence, we can define edit distance as being

$$ED(S_1, S_2) = |S_1| + |S_2| - 2 \cdot LCS(S_1, S_2),$$

with $|S_i|$ denoting the number of characters to be found in string S_i and the function $LCS(S_1, S_2)$ tells us the longest common subsequence between its two arguments.

? Which distance defined for sets does edit distance resembles the most?

Edit distance can naturally be extended with the operation of swapping a character to another one, which simply means that some

character x can be rewritten into character y in a single operation. Swapping a character is nothing else but deleting x first and inserting y, so a frequent choice is to assign a cost of 2 for such an operation. Another extension possibility of this idea is to introduce character sensitive replace costs. This way it becomes possible to penalize such character replacements more which are less likely to take place.

In order to efficiently calculate the edit distance between S_1 and S_2, each having a length of n and m characters, we can notice that for the i-long prefix of S_1 and the j-long prefix of S_2, the following recurrence holds for the edit distance that we denote by ED:

- $ED(0,j) = j, \forall j \in \{0,1,\ldots,n\}$
- $ED(i,0) = i, \forall i \in \{0,1,\ldots,m\}$

$$ED(i,j) = \min \begin{cases} ED(i-1,j)+1, \text{ for deletion} \\ ED(i,j-1)+1, \text{ for insertion} \\ ED(i-1,j-1)+cost(S_1[i],S_2[j]), \text{ for swapping} \end{cases}$$

with $cost(S_1[i],S_2[j])$ denoting the cost of directly swapping the i^{th} character of S_1 into the j^{th} character of S_2. We can assume that the cost is such that $cost(x,y) = 0$ whenever $x = y$. As mentioned earlier chosing a constant cost of 2 for any $x \neq y$ is a reasonable and frequent choice.

Example 4.6. *Determine the edit distance between $S_1 = GGCTA$ and $S_2 = AAGCTAA$ with the cost function being defined such that $cost(x,y) = 2$ for all $x \neq y$ and $cost(x,y) = 0$, otherwise.*

A	5	4	5	6	5	4	**3**	**4**
T	4	5	6	5	4	**3**	4	5
C	3	4	5	4	**3**	4	5	6
G	2	3	4	**3**	4	5	6	7
G	1	**2**	**3**	2	3	4	5	6
^	**0**	1	2	3	4	5	6	7
	^	A	A	G	C	T	A	A

Note that the characters in the longest common subsequence of S_1 and S_2 are highlighted green. Since $|S_1| = 5, |S_2| = 7$ and $LCS(S_1,S_2) = 4$, we were supposed to obtain an edit distance of $5+7-2\cdot 4 = 4$, which is exactly the number that we calculated lastly when filling out the table containing the result(s) of the (sub)problems according to the recursive problem formulation.

We shall note that there are multiple further variants of edit distance. For instance certain implementations allow for the operation of **transposition** in which two consecutive characters can be swapped

for unit cost, such that for instance $ED(bear, baer) = 1$. Obviously a transposition operation can be expressed as a composition of a deletion and an insertion operation (in either order) with a total cost of 2 units. As such, the edit distance which allows for transposition is going to assign smaller (although not strictly smaller) values for any pair of input strings.

4.6 Distances for distributions

We finally turn our attention to distances that are especially used for measuring dissimilarity between probability distributions. These distances can be used for example for feature selection [2], but they can also be used to measure the distance data points that are described by probability distributions.

[2] Euisun Choi and Chulhee Lee. Feature extraction based on the Bhattacharyya distance. *Pattern Recognition*, 36(8): 1703 – 1709, 2003. ISSN 0031-3203. DOI: https://doi.org/10.1016/S0031-3203(03)00035-9. URL http://www.sciencedirect.com/science/article/pii/S0031320303000359

The distances covered in the followings all work for both categorical and continuous distributions. Here, we discuss the variants of the distances for the discrete case only. We shall add, however, that the definitions provided next can be easily extended to the continuous case if we replace the summations by calculating integrals over the support of the random variables.

These distances can be both applied at the feature level as well as at the level of instances, i.e., the columns and rows of our data matrix, respectively. Here, we will illustrate the usage of these distances via the latter case, i.e., we would like to determine the similarity of objects that can be characterized by some categorical distribution. In the followings, we will assume that we are given with two categorical probability distributions $P(X)$ and $Q(X)$ for the random variable X.

4.6.1 Bhattacharyya and Hellinger distances

Bhattacharyya distance is defined as

$$d_B(P, Q) = -\ln BC(P, Q) \tag{4.10}$$

with $BC(P, Q)$ denoting the **Bhattacharyya coefficient**, which can be calculated according to the formula

$$BC(P, Q) = \sum_{x \in X} \sqrt{P(x)Q(x)}. \tag{4.11}$$

The Bhattacharyya coefficient is maximized when $P(x) = Q(x)$ for all $x \in X$, in which case the coefficient simply equals $\sum_{x \in X} P(x) = 1.0$. This is illustrated via the example of two Bernoulli distributions P and Q in Figure 4.9. Recall that the Bernoulli distribution is a discrete probability distribution which assigns some fixed amount of

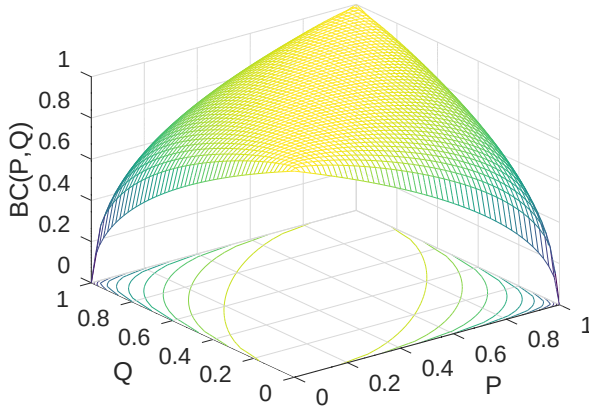


Figure 4.9: The visualization of the Bhattacharyya coefficient for a pair of Bernoulli distributions P and Q.

probability mass p for the random variable taking the value 1 and a probability mass of $1 - p$ for the random variable being equal to 0.

Bhattacharyya coefficient hence can be treated as a measure of similarity, i.e., the more similar two distributions are, the closer their Bhattacharyya coefficient is to one. This behavior of the Bhattacharyya coefficient further implies that whenever two distributions are the same, their Bhattacharyya distance is going to be equal to zero, as $\ln 1 = 0$. The Bhattacharyya distance naturally obeys symmetry which follows from the commutative nature of multiplication and summation involved in the calculation of the distance.

Triangle inequality, however, does not hold to the Bhattacharyya distance. This is because the non-linear nature of the logarithm being applied in Eq. (4.10). The term involving the logarithm severely punishes cases, when its arguments are small, i.e., close to zero, and adds an infinitesimally small penalty, whenever $P(x) \approx Q(x)$ holds.

Example 4.7. *In order to illustrate that the triangle inequality does not hold for the Bhattacharyya distance, consider the two Bernoulli distributions $P \sim Bernoulli(0.2)$ and $Q \sim Benroulli(0.6)$ that are visualized in Figure 4.10 (a). Applying the formula for the Bhattacharyya distance from Eq. (4.10), we get*

$$d_B(P, Q) = -\ln(\sqrt{0.2 * 0.6} + \sqrt{0.8 * 0.4}) = 0.092.$$

If triangle inequality hold, we would need to have

$$d_B(P, R) + d_B(R, Q) \geq d_B(P, Q)$$

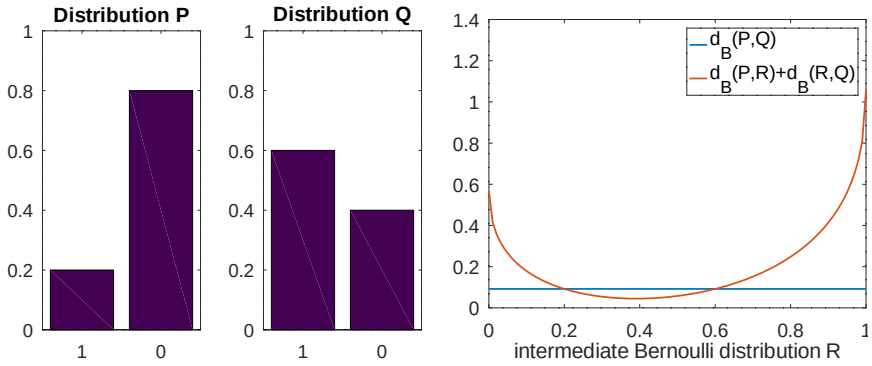
for any possible R.

However, this is clearly not the case as also depicted in Figure 4.10 (b), where we can see that it is possible to find such a distribution R that the Bhattacharyya distance $d_B(P, Q)$ exceeds the sum of Bhattacharyya distances $d_B(P, R)$ and $d_B(R, Q)$.

Indeed, the sum of Bhattacharyya distances gets minimized for $R \sim Bernoulli(0.4)$*, i.e., when* R *lies "half way" between the distributions* P *and* Q*. In that case we get*

$$d_B(P,R) + d_B(R,Q) = -\ln(\sqrt{0.2 * 0.4} + \sqrt{0.8 * 0.6}) \\ -\ln(\sqrt{0.4 * 0.6} + \sqrt{0.6 * 0.4}) = 0.045,$$

which is smaller than the previously calculated Bhattacharyya distance $d_B(P,Q) = 0.092$.



(a) Two Bernoulli distributions P and Q.

(b) The direct Bhattacharyya distance of P and Q and that as a sum introducing an intermediate distribution R.

Figure 4.10: Illustration for the Bhattacharyya distance not obeying the property of triangle inequality.

Hellinger distance is a close relative of the Bhattacharyya distance for which triangle inequality holds and which is formally defined as

$$d_H(P,Q) = \sqrt{1 - BC(P,Q)}, \tag{4.12}$$

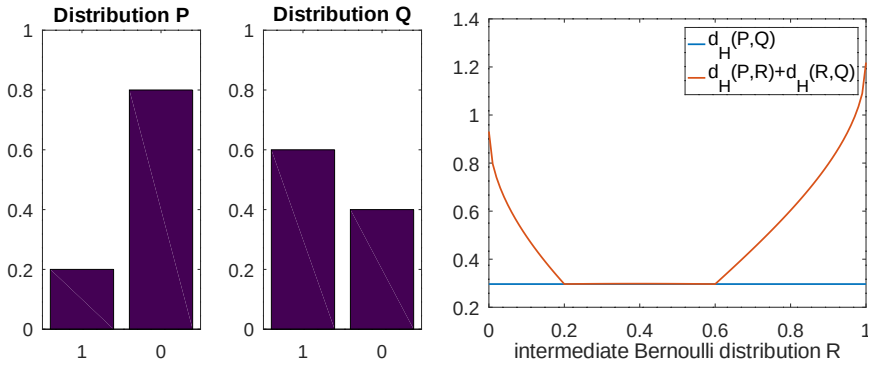
with $BC(P,Q)$ denoting the same Bhattacharyya coefficient as already defined in Eq. (4.11).

Figure 4.11 (b) illustrates via the distributions from Example 4.7 that – contrarily to the Bhattacharyya distance – the Hellinger distance fulfills the triangle inequality. Hellinger distance further differs from Bhattacharyya distance in its range, i.e., the former takes values from the interval $[0,1]$, whereas the latter takes values between zero and infinity.

An equivalent expression for the Hellinger distance reveals its relation to the Euclidean distance, i.e., it can be alternatively expressed as

$$d_H(P,Q) = \frac{1}{\sqrt{2}} \left\| \sqrt{P} - \sqrt{Q} \right\|_2, \tag{4.13}$$

treating some multivariate probability distributions of P and Q with k possible outcomes as vectors in the k-dimensional space. In order to



(a) The two Bernoulli distributions P and Q from Example 4.7.

(b) The direct Hellinger distance of P and Q and that as a sum introducing an intermediate distribution R.

Figure 4.11: Illustration for the Hellinger distance for the distributions from Example 4.7.

see the equivalence between Eq. (4.12) and Eq. (4.13), notice that

$$\left\|\sqrt{P}-\sqrt{Q}\right\|_2^2 = \left(\sqrt{P}-\sqrt{Q}\right)^\mathsf{T}\left(\sqrt{P}-\sqrt{Q}\right) \tag{4.14}$$

$$= \sqrt{P}^\mathsf{T}\sqrt{P} + \sqrt{Q}^\mathsf{T}\sqrt{Q} - 2\sqrt{P}^\mathsf{T}\sqrt{Q} \tag{4.15}$$

$$= \sum_{x\in X} P(x) + \sum_{x\in X} Q(x) - 2\sum_{x\in X}\sqrt{P(x)Q(x)} \tag{4.16}$$

$$= 2\left(1 - \sum_{x\in X}\sqrt{P(x)Q(x)}\right) \tag{4.17}$$

$$= 2\Big(1 - BC(P,Q)\Big). \tag{4.18}$$

4.6.2 *Kullback-Leibler and Jensen-Shannon divergences*

Kullback-Leibler divergence (KL divergence for short) is formally given as

$$KL(P\|Q) = \sum_{x\in X} P(x)\log\frac{P(x)}{Q(x)}.$$

Whenever $P(x) = 0$, we say that the $P(x)\log\frac{P(x)}{Q(x)}$ term in the sum cancels out, so there is no problem with the $\log 0$ in the expression.

In order the KL divergence to be defined, $Q(x) = 0$ has to imply $P(x) = 0$. Should the previous implication not hold, we cannot calculate KL divergence for the pair of distributions P and Q. Recall that a similar implication in the reverse direction is not mandatory, i.e., $P(x) = 0$ does not have to imply $Q(x) = 0$ in order the KL divergence between distributions P and Q to be quantifiable.

The previous property of KL divergence tells us that it is not a symmetric function. Indeed, not only there exist distributions P and Q for which $KL(P\|Q) \neq KL(Q\|P)$, but it is also possible that $KL(P\|Q)$ is defined, whereas $KL(Q\|P)$ is not.

It is useful to know that KL divergence can be understood as a difference between the cross-entropy of distributions P and Q and the Shannon entropy of P, denoted as

$$KL(P\|Q) = H(P;Q) - H(P).$$

Cross entropy is a similar concept to Shannon entropy introduced earlier in Section 3.3.1. Notice the slight notational difference that we employ for cross entropy ($H(P;Q)$) and joint entropy ($H(P,Q)$) discussed earlier in Section 3.3.1. Formally, cross entropy is given as

$$H(P;Q) = -\sum_{x \in X} P(x) \log Q(x),$$

which quantifies the expected surprise factor for distribution Q assuming that its possible outcomes are observed according to distribution P. In that sense, it is capable of quantifying the discrepancy between two distributions. This quantity is not symmetric in its arguments, i.e., $H(P;Q) = H(Q;P)$ does not have to hold, which again corroborates the non-symmetric nature of the KL divergence.

Cross entropy gets minimized when $P(x) = Q(x)$ holds for every x from the support of the random variable X. It can be proven by using either the **Gibbs inequality** or the **log sum inequality** that $H(P;Q) \geq H(P)$ holds for any two distributions P and Q, which implies that $KL(P\|Q) \geq 0$, with the equation being true when $P(X) = Q(X)$ for every value of X. The latter observation follows from the fact that $H(P;P) = H(P)$.

Jensen-Shannon divergence (JS divergence for short) is derived from KL divergence, additionally obeying the symmetry property. It is so closely related to KL divergence that it can essentially be expressed as an average of KL divergences as

$$JS(P,Q) = \frac{1}{2}\left(KL(P\|M) + KL(Q\|M)\right),$$

with M denoting the average distribution of P and Q, i.e.,

$$M = \frac{1}{2}\left(P + Q\right).$$

Example 4.8. *Imagine Jack spends 75%, 24% and 1% of his spare time reading some book, going to the cinema and hiking. Two of Jack's colleagues – Mary and Paul – devote their spare time according to the probability distributions* $[0.60, 0.30, 0.10]$ *and* $[0.55, 0.40, 0.05]$ *for the same activities. Let us quantify which colleague of Jack likes to spend their spare time the least dissimilar to Jack, i.e., whose distribution lies the closest to that of Jack's.*

For measuring the distances between the distributions, let us make use of the different distances that we covered in this chapter, including the ones that are not specially designed to be used for probability distributions.

The results of the calculations are included in Table 4.1 and the code snippet written in vectorized style that could be used to reproduce the results can be found in Figure 4.13. A crucial thing to notice is that depending which notion of distance we rely on, we arrive to different answers regarding which colleague of Jack has a more similar preference for spending their sparse time. What this implies that data mining algorithms which rely on some notion of distance might result in different outputs if we modify how we define the distance between the data points.

The penultimate row of Table 4.1 and the one before it also highlights the non-symmetric nature of KL divergence.

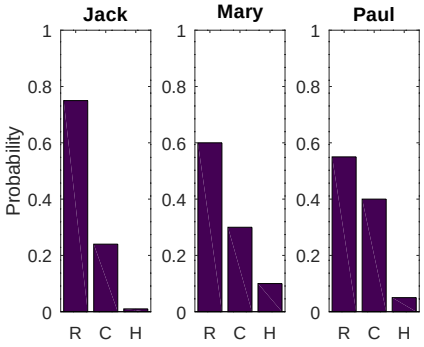


Figure 4.12: Visualization of the distributions regarding the spare time activities in Example 4.8. R, C and H along the x-axis refers to the activities reading, going to the cinema and hiking, respectively.

	Mary	Paul
city block distance	**0.300**	0.400
Euclidean distance	**0.185**	0.259
Chebyshev distance	**0.150**	0.200
cosine distance	**0.204**	0.324
Bhattacharyya distance	0.030	**0.026**
Hellinger distance	0.171	**0.160**
$KL(Jack\|\|colleague)$	**0.091**	0.094
$KL(colleague\|\|Jack)$	0.163	**0.114**
JS divergence	0.027	**0.025**

Table 4.1: Various distances between the distribution regarding Jack's leisure activities and his colleagues. The smaller values for each notions of distances is highlighted in bold.

4.7 *Euclidean versus non-Euclidean distances*

In the previous sections, we have familiarized with a series of distance concepts. Some of them were measuring the "actual, physical" distance between data points in a representation where taking the average of two points was possible and made sense. Such distances are referred to as Euclidean distances. There were, however, other distances which operated over objects the average of which could not be easily interpreted or defined. This other group of distances form the family of non-Euclidean distances.

? Try to categorize the various distances discussed over the chapter whether they belong to the Euclidean or the non-Euclidean family of distances.

CODE SNIPPET

```
j = [0.75 0.24 0.01]; %distribution for Jack
% distributions for Jack's colleagues
Colls = [0.60 0.30 0.10; 0.55 0.40 0.05];

e_d = norm(j-Colls, 'rows')';      % Euclidean distances
cb_d = norm(j-Colls, 1, 'rows')';  % city block distances
c_d = norm(j-Colls, Inf, 'rows')'; % Chebyshev distances

normalized_j = j / norm(j);
normalized_Colls = Colls ./ norm(Colls, 'rows');
cosine_distances = acos(normalized_j * normalized_Colls');

BC = sqrt(j)*sqrt(Colls)'; % Bhattacharyya coefficient
bd = -log(BC);             % Bhattacharyya distance
hd = sqrt(1-BC);           % Hellinger distance

M = 0.5 * (j+Colls);
KL = @(P,Q) sum(P.*log(P./Q), 2)
kl_forward = KL(j, Colls);
kl_backward = KL(Colls, j);
js = 0.5 * (KL(j, M) + KL(Colls, M));
```

Figure 4.13: Sample code to calculate the difference distance/divergence values for the distributions from Example 4.8.

4.8 Summary of the chapter

This chapter introduced the concepts of distances and similarities between data points and introduced a series of distances frequently applied in data mining approaches. Readers of this chapter are expected to know the different characteristics of the different distances and to be able to argue for a particular choice of distance upon dealing with a dataset.

5 | Finding Similar Objects Efficiently

Learning Objectives:
- Locality sensitive hashing (LSH)
- AND/OR constructions for LSH
- Bloom filters

Readers of this chapter can learn about different ways of quantifying similarity between pairs of objects. In particular, by the end of the chapter one should

- understand and argue for the need of approximate methods to measure for the similarity of objects,
- assess these approximate methods for measuring similarity from a probabilistic perspective.

5.1 *Locality Sensitive Hashing*

Supposed we have n elements in our dataset, finding the item which is the most similar to every item in the dataset takes $\binom{n}{2} = O(n^2)$ comparisons. For large values of n, this amount of comparison is definitely beyond what is practically manageable.

What we would like to do, nonetheless, is to form roughly equally-sized bins of items in such a way that similar items make it to the same bin with high probability, while dissimilar points are assigned to the same bin infrequently with low probability. Naturally, we want to obtain this partitioning of the items without the need to calculate all (or even most) of the pairwise distances. If we manage to come up with such a partitioning mechanism, then it obviously suffices to look for points of high similarity within the same bin for every points.

Example 5.1. *Suppose you want to find the most similar pairs of book in a library consisting of* 10^6 *books. In order to do so, you can perform* $\binom{10^6}{2} \approx 5 \cdot 10^{11}$ *(500 billion) comparisons if you decide to systematically compare books in every possible way. This is just too much expense.*

If you manage to partition the data into, say, 100 equal parts in a way that it is reasonable to assume that the most similar pairs of books are assigned within the same partition, then you might consider performing a reasonably reduced number of pairwise comparisons, i.e., $100 \cdot \binom{10^4}{2} \approx 5 \cdot 10^9$.

As the above example illustrated, the general rule of thumb is that if we are able to divide our data into k (roughly) equally-sized partitions and restrict the pairwise comparisons conducted to only those pairs of objects which are grouped to the same partition, a k-fold speedup can be achieved. The question still remains though, how to partition our object in a reasonable manner. We shall deal with this topic next for the Jaccard distance first.

5.1.1 *Minhash functions*

In this setting, we have a collection of data points that can be described by sets of attributes characterizing them. As an example, we might have documents being described by the collection of words which they include or restaurants having a particular set of features and properties (e.g. expensive, children friendly, offering free parking). We can organize these binary relations in a **characteristic matrix**. Values in a characteristic matrix tell us whether a certain object (represented as a row) has a certain property (represented by columns). Ones and zeros in the characteristic matrix represent the presence and absence of properties, respectively.

CODE SNIPPET

```
C=[1 0 0 1; 0 0 1 0; 1 1 1 0; 0 1 1 1; 0 0 1 0; 1 0 1 0];
>> C =

   1   0   0   1
   0   0   1   0
   1   1   1   0
   0   1   1   1
   0   0   1   0
   1   0   1   0
```

Figure 5.1: Creating the characteristic matrix of the sets

In Figure 5.2 we illustrate how can we permute the rows of the characteristic matrix in Octave. The method `randperm` creates us a random ordering of the integers that we can use for reordering our matrix. The way permutation `[6 1 3 4 2 5]` can be interpreted is that the first row of the reordered matrix will contain the sixth row from the original matrix, the second row of the reordered matrix will consist of the first row of the original matrix and so on.

The following step we need to perform is to determine the position of the first non-zero entry in the permuted characteristic matrix, which is essentially the definition of the **minhash value** for a set given some random reordering of the characteristic matrix.

CODE SNIPPET

```
rand('seed', 1)  % ensure repoducibility of the permutation
% create a permutation vector with the size of the
% the number of rows in the characteristic matrix
idx = randperm(size(C,1))
>> idx =
   6   1   3   4   2   5

C(idx, :)  % let's see the results of the permutation
>> ans =
   1   0   1   0
   1   0   0   1
   1   1   1   0
   0   1   1   1
   0   0   1   0
   0   0   1   0
```

Figure 5.2: Creating a random permutation of the characteristic matrix

In our Octave code we will make use of the fact that the elements we are looking for are known to be the maximal elements in the characteristic matrix. Hence, we can safely use the `max` function which is capable of returning not only the column-wise maxima of its input matrix, but their location as well. Conveniently, this function returns the first occurrence of the maximum value if it is present multiple times, which is exactly what we need for determining the minhash value of a set for a given permutation of the characteristic matrix.

In Figure 5.3 we can see that invoking the `max` function returns us all ones for the `max_values`. This is unsurprising as the matrix in which we are looking for the maximal values consist of only zeros and ones. More importantly, the `max_indices` variable tells us the location of the first occurrences of the ones in the matrix for every column. We can regard every element of this output as the minhash function value for the corresponding set in the characteristic matrix given the current permutation of its rows.

CODE SNIPPET

```
[max_values, max_indices] = max(C(idx, :))
>> max_values =
   1   1   1   1

max_indices =
   1   3   1   2
```

Figure 5.3: Determine the minhash function for the characteristic matrix

We should add that in a real world implementation of minhashing – when we have to deal with large-scale datasets – we do not explicitly store the characteristic matrix in a dense matrix format as it was provided in the illustrative example.

The reason is that since most of the entries of the characteristic matrix are typically zeros, it would be hugely wasteful to store them as well. Instead, the typical solution is to store the characteristic matrix in a sparse format, where we only need to store the non-zero indices for every row. Ideally and most frequently, the amount of memory consumed for storing the indices for the non-zero indices take orders of magnitude less compared to the explicit full matrix representation.

The other practical observation we should make is that actually swapping rows of the characteristic matrix when performing permutation is another source of inefficiency. Instead, what we should do is to leave the characteristic matrix intact and generate an *alias identifier* for each row. This alias identifier acts as a virtual row index for a given row. From the running example above, even though the first row of the characteristic matrix would stay in its original position, we would treat this row as if it was row number 6, which comes from the permutation indices we just generated.

Initially, we set the minhash value for every set to be infinity. Then upon investigating a particular row of the characteristic matrix, whichever set contains a one for that given row, we update their corresponding minhash values found so far by the minimum of the alias row identifier of the current row and the current minhash value of the given set. After processing every row of the (virtually) permuted characteristic matrix, we would be given with the correct minhash values for all of the sets represented in our characteristic matrix.

Example 5.2. *Let us derive how would the efficient calculation of the minhash values look like from iteration to iteration for the sets in the characteristic matrix given by*

$$C = \begin{array}{c} \\ 1(2) \\ 2(5) \\ 3(3) \\ 4(4) \\ 5(6) \\ 6(1) \end{array} \begin{array}{c} \begin{array}{cccc} S_1 & S_2 & S_3 & S_4 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}.$$

As for the permutation of the rows, we used the same permutation vector as before, i.e., $[6, 1, 3, 4, 2, 5]$*, meaning that the rows of the characteristic matrix in their original order will function as the second, fifth, third, fourth, sixth*

Algorithm 1: Efficient calculation of the minhash values for multiple sets stored in characteristic matrix C and a given permutation of rows π.

Require: Characteristic matrix $C \in \{0,1\}^{k \times l}$ and its permutation of row indices π.
Ensure: minhash vector $\boldsymbol{m}$

```
1: function CalculateMinhashValue(C, π)
2:     m ← [∞]^l
3:     for row_index=1 to k do
4:         virtual_row_index ← π[row_index]
5:         for set_index=1 to l do
6:             if C[row_index, set_index] == 1 then
7:                 m[set_index] ← min(m[set_index], virtual_row_index)
8:             end if
9:         end for
10:    end for
11:    return m
12: end function
```

and first rows in the (virtually) permuted matrix, respectively. These virtual row indices will be denoted by the row aliases which are put in front of every row of the characteristic matrix for convenience in a parenthesis right after the original row numbers.

Table 5.1: Step-by-step derivation of the calculation of minhash values for a fixed permutation.

Iteration	row alias	S_1	S_2	S_3	S_4
0		∞	∞	∞	∞
1	2	$2 = \min(2,\infty)$	∞	∞	$2 = \min(2,\infty)$
2	5	2	∞	$5 = \min(5,\infty)$	2
3	3	$2 = \min(3,2)$	$3 = \min(3,\infty)$	$3 = \min(3,5)$	2
4	4	2	$3 = \min(4,3)$	$3 = \min(4,3)$	$2 = \min(4,2)$
5	6	2	3	$3 = \min(6,3)$	2
6	1	$1 = \min(1,2)$	3	$1 = \min(1,3)$	2

After the last row is processed, we obtain the correct minhash values for sets S_1, S_2, S_3, S_4 as 1, 3, 1, 2, respectively. These minhash values can be conveniently read off from the last row of Table 5.1 and they are identical with the Octave-based calculation included in Figure 5.4.

Now that we know how to calculate the minhash value of a set with respect to a given permutation of the characteristic matrix, let us introduce the concept of **minhash signature**s. A minhash signature is nothing else but a series of minhash values stacked together according to different (virtual) permutations of the characteristic matrix.

? Before checking out the next code snippet, could you extend the previous code snippets in order to support the creation of minhash signatures?

An important property of minhash signatures is that for any pair of sets (A, B), we get that the relative proportion of times their minhash values match each other is going to be equal to their Jaccard similarity j, given that we perform all the possible permutations of

CODE SNIPPET

```
signature = zeros(8, size(C,2));
for k=1:size(signature, 1)
  [max_values, max_indices] = max(C(randperm(size(C,1)),:));
  signature(k,:) = max_indices;
end

>> signature =
   1   3   1   2
   1   4   1   3
   1   1   1   2
   2   4   1   3
   3   2   1   2
   1   1   1   2
   1   1   1   3
   2   1   1   1
```

Figure 5.4: Determine the minhash function for the characteristic matrix

the elements of the sets. We never make use of this result directly, since performing every possible permutations would be computationally prohibitive. A more practical view of the same property is that assuming that we generated just a single random permutation of the elements of the sets, the probability that the two sets will end up having the same minhash value exactly equals their actual Jaccard similarity j, i.e.,

$$P(h(A) = h(B)) = j. \tag{5.1}$$

Assume that $|A \cap B| = m$ and $|A \triangle B| = n$ holds, i.e., the intersection and the symmetric difference between the two sets are m and n, respectively. From here, it also follows that $|A \cup B| = n + m$. This means that there are $n + m$ rows in the characteristic matrix of sets A and B which can potentially influence the minhash value of the two sets, constituting $(n + m)!$ possible permutation possibilities in total. If we now consider the relative proportion of those permutations which result in such a case when one of the m elements in the intersection of the two sets precede any of the remaining $(n + m - 1)$ elements in a random permutation, we get

$$\frac{m(n+m-1)!}{(n+m)!} = \frac{m}{(n+m)} = \frac{|A \cap B|}{|A \cup B|} = j.$$

5.1.2 *Analysis of LSH*

In order to analyze locality sensitive hashing from a probabilistic perspective, we need to introduce some definitions and notations

first.

We say that some hash function h is a member of the (d_1, d_2, p_1, p_2)-sensitive hash functions, if for any pair of points (A, B) the following properties hold

1. $P(h(A) = h(B)) \geq p_1$, whenever $d(A, B) < d_1$
2. $P(h(A) = h(B)) \leq p_2$, whenever $d(A, B) > d_2$.

What this definition really requires is that whenever the distance between a pair of points is substantially low (or conversely, whenever their high), then their probability of being assigned with the same hash value should also be high. Likewise, if their pairwise distance if substantially large, their probability of being assigned to the same bucket should not be large.

Example 5.3. *Based on that definition and our previous discussion on the properties of the Jaccard similarity (hence the distance as well), we can conclude that the minhash function belongs to the family of* $(0.2, 0.4, 0.8, 0.6)$*-sensitive functions.*

Indeed, any set with a Jaccard distance at most 0.2 *(which equals a Jaccard similarity of at least* 0.8*), the probability that they will receive the same minhash function for some random permutations is at least* 0.8*. Likewise, for any pairs of sets with a Jaccard distance larger that* 0.4 *(equivalently with their Jaccard similarity not exceeding* 0.6*), the probability of assigning them identical minhash value is below* 0.6 *as well.*

If we have a simple locality sensitive function belonging to the family of hash functions of certain sensitivity, we can create composite hash functions based on them. A good idea to do so might be to calculate a series of hash functions first, just like we did it for minhash signatures.

Now instead of thinking of such a minhash signature of length k – calculated based on k independent permutation of the characteristic matrix – let us treat these signatures as b independent bands, each consisting of r minhash values, hence $k = rb$. A sensible way to combine the k independent minhash values could be as follows: investigate every band of minhash values and align two objects into the same bucket if there exists at least one band of minhash values over which they are identical to each other for all the r rows of a band. This strategy is illustrated by Figure 5.5. According to the LSH philosophy, when searching for pairs of objects of high similarity, it suffices to look for such pairs within each of the buckets, since we are assigning objects to buckets in a way that similar objects will tend to be assigned into the same buckets.

Let us now investigate the probability of assigning two objects with some Jaccard similarity j into the same bucket when employing

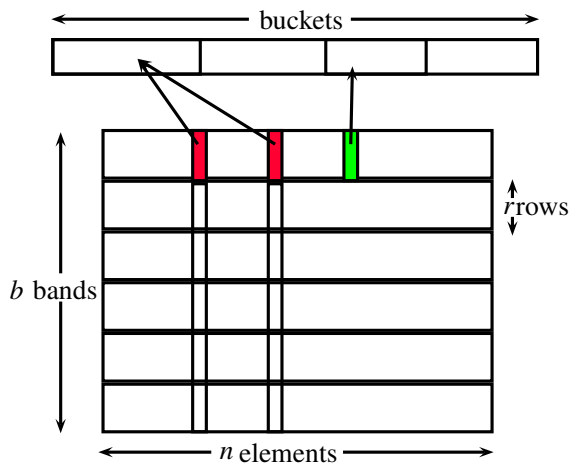


Figure 5.5: Illustration of Locality Sensitive Hashing. The two bands colored red indicates that the minhash for those bands are element-wise identical.

b bands of r rows in the above described manner. Recall that due to our previous observations related to the Jaccard similarity, when $b = r = 1$ we simply get that the results of Eq. (5.1). The question is now, how shall we modify our expectations in the general case, when both b and r are allowed to differ from 1.

Observe first that the probability of obtaining the exact same minhash values for a pair of objects over r random permutations is simply j^r, that is

$$P(h_1(A) = h_1(B), h_2(A) = h_2(B), \ldots, h_r(A) = h_r(B)) = j^r. \tag{5.2}$$

From Eq. (5.2), we get that the probability of having at least one mismatch between the pairwise minhash values of the two objects over r permutations of the characteristic matrix equals $1 - j^r$.

Finally, we can notice that observing *at least one* band of identical minhash values out of b bands for a pair of objects is just the complement event of observing only such bands that mismatch on at least one position. This means that our final probability of assigning two objects into the same bucket in the general case, when we treat minhash signatures of length $k = rb$ as b independent bands of r minhash values can be expressed as

$$1 - (1 - j^r)^b. \tag{5.3}$$

As we can see from Eq. (5.3), r and b has opposite effects for the probability of assigning a pair of objects to the same bucket. Increasing b increases the probability for a pair of sets of becoming a candidate that we shall check for their actual similarity, whereas increasing r decreases the very same probability. When we increase b, we are essentially allowing for more trials to see an identical band of r minhash values. On the other hand, when we increase r, we are

imposing a more stringent condition on when a band is regarded identical. Hence additional rows within a band act in a restrictive manner, whereas additional bands have a permissive effect when assigning objects to bands.

Notice how the previously found probability can be motivated by a coin toss analogy. Suppose we are tossing a possibly biased coin (that is the probability of tossing a head and a tail are not necessarily 0.5 each) and tossing a head is regarded as our lucky toss. Now the probability we are interested corresponds to the probability of tossing at least one head out of b tosses for which the probability of tossing a tail equals $1 - j^r$. This probability can be exactly given by Eq. (5.3).

? Each subfigure within Figure 5.6 corresponds to a different choice of $b \in \{1, 2, \ldots, 10\}$. Can you argue which curve corresponds to which value of b?

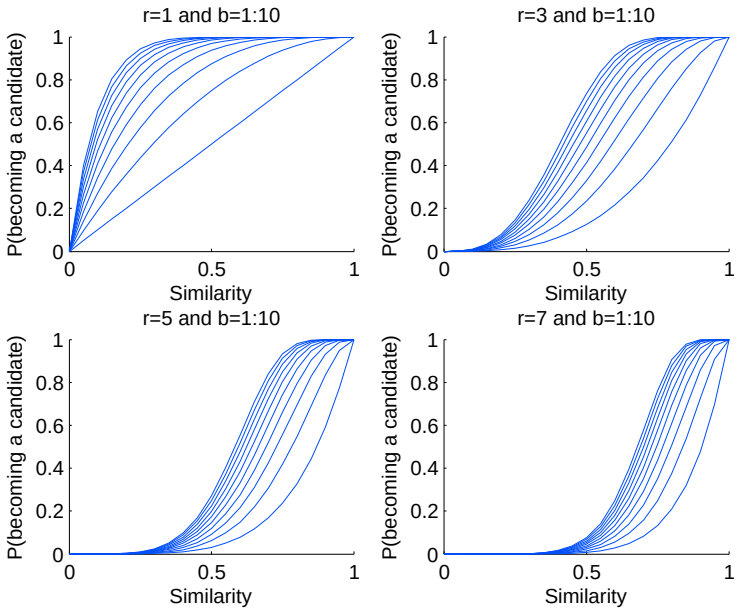


Figure 5.6: Illustration of the effects of choosing different band and row number during Locality Sensitive Hashing for the Jaccard distance. Each subfigure has the number of rows per band fixed to one of $\{1, 3, 5, 7\}$ and the values for b range in the interval $[1, 10]$.

Figure 5.6 visualizes different values of Eq. (5.3) as a function of varying b and r values. Notice that for the top-left subfigure (with $r = 1$), only the permissive component of the formula takes its effect, which results in extreme lenience towards data points being assigned to the same bucket even when their true Jaccard similarity found along the x-axis is infinitesimal. The higher value we pick for r, the more pronouncedly this phenomenon can be observed.

? Can you make an argument which curves of Figure 5.6 are reasonable to be compared with each other? (Hint: Your suggestion for which curves are comparable with each other might span across multiple subplots.)

We now have a clear way for determining the probability for two sets with a particular Jaccard similarity to be assigned to the same bucket, hence to become such a pair of candidates that we regard to be worthwhile for investigating their exact similarity. Assume that we have some application, where we can determine some similarity threshold J, say, 0.6 which corresponds to such a value above which we would like to see pairs of sets to be assigned to the same bucket.

With the hypothetical threshold of 0.6. in mind, Figure 5.7 depicts the kind of probability distribution we would like to see for assigning a pair of objects to the same bucket as a function of their actual (Jaccard) similarity.

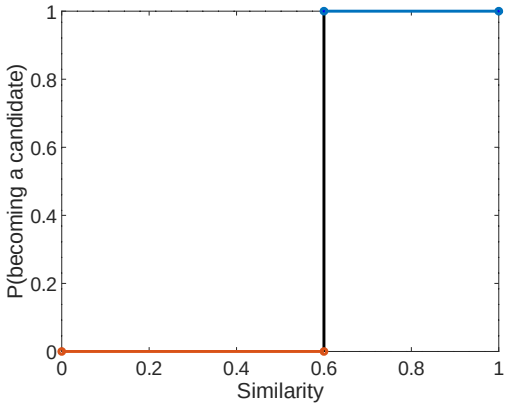
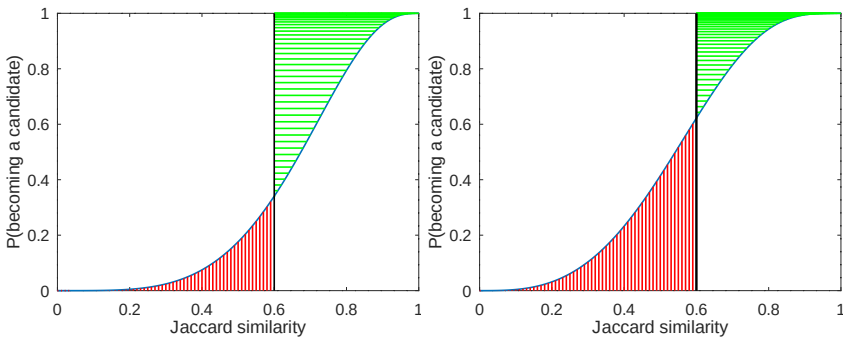


Figure 5.7: The ideal behavior of our hash function with respect its probability for assigning objects to the same bucket, if our threshold of critical similarity is set to 0.6.

Unfortunately, we will never be able to come up with a hash function which would behave as a discontinuous step function similar to the idealistic plot in Figure 5.7. Our hash function will, however, surely assign at least a tiny non-zero probability mass for pairs of objects being assigned to the same bucket even in such cases, when it is otherwise not desired, i.e., their similarity falls behind the expected threshold J. This kind of error is what we call the **false positive error**, since we are erroneously treating dissimilar object pairs positively by assigning them into the same bucket. The other possible error is the **false negative error** when we fail to assign substantially similar pairs of objects into the same bucket. The false positive and false negative error region is marked by red vertical and green horizontal stripes in Figure 5.8, respectively.



(a) Applying 3 bands and 4 rows per band. (b) Applying 4 bands and 3 rows per band.

Figure 5.8: The probability of assigning a pair of objects to the same basket with different number of bands and rows per bands.

Example 5.4. *Figure 5.8 (a) and (b) illustrates the probability for assigning a pair of objects to the same bucket when 12 element minhash signa-*

tures are treated as a composition of 4 bands and 3 rows within a band (Figure 5.8 (a)) and 3 bands and 4 rows within a band (Figure 5.8 (b)). Note how different the sizes are for the areas striped by red and green. This demonstrates how one can control for the false positive and false negative rate by changing the values of b and r.

There will always be a tradeoff between the two quantities, in the sense that we can decrease one kind of the error rates at the expense of increasing the other kind and vice versa. Intuitively, if we wish to reduce the amount of false negative error, we need to be more permissive for assigning pairs of objects to the same bucket. Becoming more permissive, however, will not only increase the probability for truly similar pairs of points being assigned to the same bucket only, but potentially for dissimilar ones as well.

Assume that we have r independent hash functions $h_1, h_2, \ldots, h_r$, each being a member of the family of (d_1, d_2, p_1, p_2)-sensitive hash functions. If we construct a composit hash function h' which assigns identical hash value for a pair of objects if they receive the same atomic hash values for *all* of the h_i, $1 \leq i \leq r$ independent (d_1, d_2, p_1, p_2)-sensitive hash functions, then the h' hash function belongs to the family of (d_1, d_2, p_1^r, p_2^r)-sensitive hash functions.

Supposing now that we have b independently trained hash functions each belonging to the family of (d_1, d_2, p_1, p_2)-sensitive hash functions. By creating a composite hash function h' which assigns two objects the same hash value if they are assigned the same hash value by *either* of $h_1, h_2, \ldots, h_b$, the resulting hash function h' will belong to the family of $(d_1, d_2, 1-(1-p_1)^b, 1-(1-p_2)^b)$-sensitive hash functions.

AND/OR-amplifications can be regarded as a generalization of the kind of sensitivity analysis we have conducted so far. Note that AND/OR-amplifications can be nested in an arbitrarily complex manner and order. According to our original idea, we proposed treating our minhash signature as a series of AND-amplified atomic hash functions, for the outputs of which we applied OR-amplifications.

By introducing a general characterization on the effects towards the sensitivity of composit hash functions, we can easily analyze any complex composit hash functions as illustrated by the following example.

? Our original idea was to perform AND-constructions of the atomic hash functions first, followed by OR-constructions. What would be the rule for assigning a pair of objects into the same bucket if the two different kinds of amplifications were performed in reverse order (i.e., OR-amplifications followed by AND-amplifications)?

Example 5.5. *We have argued previously that the minhash function belongs to the family of* $(0.2, 0.6, 0.8, 0.4)$*-sensitive hash functions. In fact, it is true that the minhash function is a* $(d_1, d_2, 1-d_1, 1-d_2)$*-sensitive function for any choice of* $0 \leq d_1 < d_2 \leq 1$.

What would be the sensitivity of the composite hash function that we obtain by applying AND*-construction with* $r = 3$ *followed by an* OR*-construction using* $b = 4$*?*

The composite hash function that we derived in the previous way from a $(0.2, 0.6, 0.8, 0.4)$*-sensitive atomic hash function belongs to the family of* $(0.2, 0.6, 0.943, 0.232)$*-sensitive hash functions. The* p_1 *sensitivity of the composite hash function can be simply derived as*

$$0.943 = 1 - (1 - 0.8^3)^4,$$

The value for p_2 *can be calculated in a similar manner, by replacing* 0.8 *with* 0.4*.*

Figure 5.9 depicts the behavior of the LSH approach with respect the above specified parameters towards the probability of assigning pairs of objects to the same bucket. Note that it would also be possible to perform the different compositions in the reverse order, the results of which is also included in Figure 5.9.

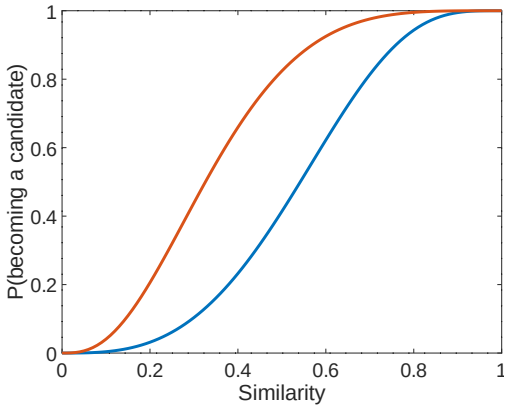


Figure 5.9: The probability curves when performing AND-construction ($r = 3$) followed by OR-constructions ($b = 4$). The curves differ in the order the AND,-and OR-constructions follow each other.

CODE SNIPPET

```
a=@(p, s) p.^s % responsible for AND-constructions
o=@(p, b) 1-(1-p).^b % responsible for OR-constructions
j=0:0.05:1;  % the range of Jaccard similarity
plot(j, o(a(j, 3), 4), j, a(o(j, 4), 3))
```

? Can you argue which curve in Figure 5.9 corresponds to performing AND-amplifications followed by OR-constructions? If clueless, try invoking the code snippet in Figure 5.10.

Figure 5.10: Illustration of combining AND/OR-amplifications in different order.

5.1.3 LSH for cosine distance

We have seen previously a locality sensitive hash function for the Jaccard distance. A natural question to ask ourselves, how can we provide similarly nice hash functions for distances of different nature

beyond the Jaccard distance. We now turn our attention to the cosine distance and come up with a hash function for it.

The idea behind the hash function used for cosine distance is based on projecting vectors onto randomly drawn hyperplanes. Figure 5.11 provides a brief sketch regarding the relation between the dot product and projections of vectors onto hyperplanes, which can be safely skipped by readers who feeling themselves familiar enough about it.

MATH REVIEW | DOT PRODUCT

The **dot product** (or **inner product**) between tho vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^d$ is simply the sum of their coordinatewise product,

$$\boldsymbol{x}^\intercal \boldsymbol{y} = \sum_{i=1}^{d} x_i y_i. \tag{5.4}$$

Dot product occur at many important places, e.g., as we saw it in Section 4.3, it is related to the cosine of the angle enclosed by two vectors. Additionally, dot product is related to the perpendicular projection of vector $\boldsymbol{x}$ onto vector $\boldsymbol{y}$, which is given by

$$\boldsymbol{x}_{\perp y} = \frac{\boldsymbol{x}^\intercal \boldsymbol{y}}{\boldsymbol{y}^\intercal \boldsymbol{y}} \boldsymbol{y}. \tag{5.5}$$

The way we can interpret Eq. (5.5) is that the orthogonal projection of $\boldsymbol{x}$ onto $\boldsymbol{y}$ points to the orientation of $\boldsymbol{y}$, and its length equals that of the dot product of the two vectors.
A slightly different view of the dot product hence is the following: it equals the *signed* distance of vector $\boldsymbol{x}$ from the hyperplane which is normal to $\boldsymbol{y}$. Now the sign of dot product tells us on which 'side' of the hyperplane defined by $\boldsymbol{y}$ vector $\boldsymbol{x}$ is located at. If the dot product is positive, it means that $\boldsymbol{x}$ lies in the positive half-space. Likewise, when it is negative we can conclude that $\boldsymbol{x}$ is located at the negative half-space. Finally, if the dot product is zero then $\boldsymbol{x}$ is exactly positioned on the hyperplane to which $\boldsymbol{y}$ is normal to.

Figure 5.11: Dot product

We need to construct next an analogous counterpart of the min-hash signatures that we were relying on for the Jaccard distance. If we can do so, we can pretty much apply all the analysis that we previously gave for it in terms of false positive and negative rates. Let us define for the case of cosine similarity a singleton hash function which takes as input some $\boldsymbol{x} \in \mathbb{R}^d$ as

$$h_s(\boldsymbol{x}) = sign(\boldsymbol{s}^\intercal \boldsymbol{x}), \tag{5.6}$$

with *sign* referring to the sign function and $\boldsymbol{s} \in \mathbb{R}^d$ being a randomly

sampled normal vector to some hyperplane in $\mathbb{R}^d$.

When applying one such hash function to a pair of vectors $(\boldsymbol{x}, \boldsymbol{y})$, then the probability for the two vectors receiving the same hash value introduced in Eq. (5.6) equals

$$P(h_s(\boldsymbol{x}) = h_s(\boldsymbol{y})) = \frac{180 - \theta}{180}, \tag{5.7}$$

where θ denotes the angle enclosed by vectors $\boldsymbol{x}$ and $\boldsymbol{y}$.

Properties of the dot product discussed in Figure 5.11 explains why this **geometric probability** holds. The larger the angle between two vectors, the more likely that a randomly drawn hyperplane could behave as an 'intruder', i.e., lie in between the two vectors. Similarly, if the angle enclosed by a pair of vectors is small, then the chances for drawing such a hyperplane which splits the space such that the two vectors are located at the different **half-space**s is also small.

Figure 5.12 nicely illustrates that the probability of drawing such a hyperplane which separates the two endpoints of vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is proportional to the angle enclosed by them, or equivalently, the probability of the two vectors not to be separated is proportional to $180 - \theta$ as stated in Eq. (5.7).

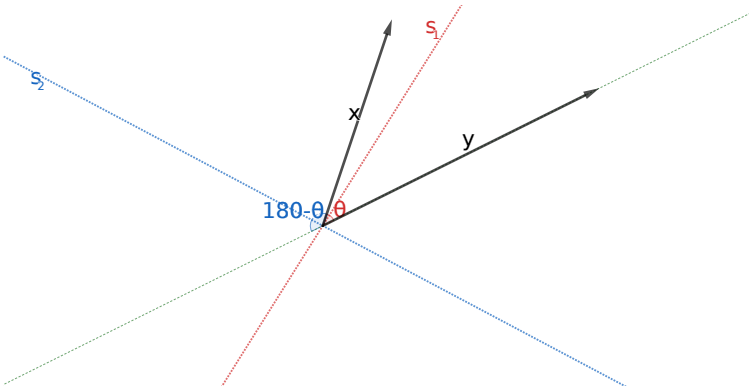


Figure 5.12: Illustration of the geometric probability defined in Eq. (5.7). Randomly drawn hyperplanes s_1 and s_2 are denoted by dashed lines. The range of the separating angle and one such hyperplane (s_1) is drawn in red, whereas the non-separating range of angle and one such hyperplane (s_2) are in blue.

Based on the previous observations, we can conclude that the hash function introduced in Eq. (5.6) belongs to the family of hash functions with $(d1, d2, (180 - d1)/180, (180 - d2)/180)$ sensitivity. In order to get a similar construction to minhash signatures all we need to do is to calculate and concatenate the hash function defined in Eq. (5.6) k times, by generating a different random hyperplane to project the data points onto. Such vectors of concatenated hash functions are often referred to as **sketch**es. The following example illustrates how accurately the relative proportion of sketch values can help us infer an approximation towards the angle enclosed by a pair of vectors as a function of the number of the random hyperplanes to project vectors onto.

Example 5.6. *Suppose we have two 3-dimensional vectors*

$$\boldsymbol{x} = \begin{bmatrix} 2 \\ -1 \\ 2 \end{bmatrix}, \boldsymbol{y} = \begin{bmatrix} 4 \\ 1 \\ 3 \end{bmatrix}.$$

According to Chapter 4.3, we know that their true angle can be obtained as

$$acos\left(\frac{\boldsymbol{x}^\intercal \boldsymbol{y}}{\|\boldsymbol{x}\|_2 |\boldsymbol{y}\|_2}\right),$$

the calculation of which tells us that the angle θ enclosed by $\boldsymbol{x}$ and $\boldsymbol{y}$ equals 30.81 degrees.

Figure 5.13 demonstrates how accurate the approximation towards to true angle between the two vectors gets as a function of the number of hyperplanes the vectors are projected onto. It can be noticed that the approximation for the true angle between the vectors – that we derive from the relative proportion of times their hash functions according to different random projections as defined in Eq. (5.6) are the same – converges to the actual angle enclosed by them as we increase the number of hyperplanes the vectors are projected onto.

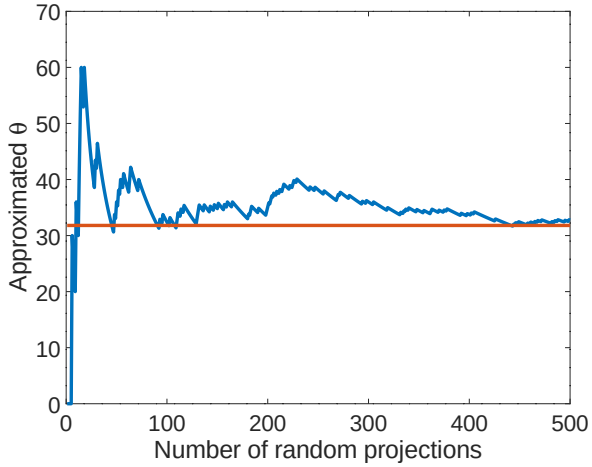


Figure 5.13: Approximation of the angle enclosed by the two vectors as a function of the random projections employed. The value of the actual angle is represented by the horizontal line.

In practice the kind of vectors $\boldsymbol{s}$ that are involved in calculating the hash value of vectors are often chosen to consist of values either $+1$ or -1. That is a frequent choice for $\boldsymbol{s}$ is $\boldsymbol{s} \in \{+1, -1\}^d$, which is convenient as this way calculating their dot product with any vector $\boldsymbol{x}$ simplifies to taking a (signed) sum of the elements of vector $\boldsymbol{x}$.

5.2 *Approaches disallowing false negatives*

As our previous discussion highlighted, locality sensitive hashing would always produce a certain fraction of object pairs which have

a high degree of similarity, nonetheless fail to get assigned into the same bucket. Likewise, there are going to be object pairs that are spuriously regarded as being similar by the algorithm, despite having a large distance between them. There are certain kinds of applications, where the former kind of problems, i.e., false negative errors are simply intolerable.

One domain where false negatives could be highly undesired is related to criminal investigations. Suppose police investigators found the fingerprints of a burglar. When looking for suspects, i.e., people whose fingerprint has a high degree of similarity to the one left behind at the crime scene, it would be inacceptable not to identify the actual perpetrator as a suspect.

? Can you think of further scenarios where false negative errors are highly undesirable?

It seems that if we want to exclude the possibility of false negatives, we certainly have to calculate the actual similarity between all the pairs of objects in a systematic manner. Luckily, there exist certain heuristics-based algorithms that can eliminate calculating the exact similarity between certain pairs, yet assure that there will be no false negatives. Suppose we have some similarity threshold S in our mind, such that we would like to surely find those pairs of objects for which their similarity is at least S.

The general idea behind these approaches is that for a pair of objects (x, y), we shall introduce a *fast-to-calculate* upper bound (s') on their actual similarity (s), i.e., $s'(x, y) > s(x, y)$. Whenever our quickly calculable upper bound fails to surpass the desired minimum amount of similarity, that is $s'(x, y) < S$, we can be absolutely sure that we do not risk anything by omitting the calculation of $s(x, y)$, i.e., the actual similarity between the object pair (x, y).

5.2.1 *Length based filtering*

Assume that we are calculating the similarity between pairs of sets according to Jaccard similarity. We can (over)estimate the true Jaccard similarity for any pair of sets of cardinalities m and n with $m \leq n$ by taking the fraction $\frac{m}{n}$. This is true as even if the smaller set is a proper subset of the larger set, the size of their intersection equals the number of elements in the smaller set (m), whereas the cardinality of their intersection would equal the size of the larger set (n). Now assuming that $\frac{m}{n}$ is below some predefined threshold S, it is pointless to actually investigate the true Jaccard similarity as it is also definitely below S.

Example 5.7. *We have two sets* $A = \{2, 1, 4, 7\}$ *and* $B = \{3, 1, 2, 6, 5, 8\}$. *This means that* $m = 4$ *and* $n = 6$, *which implies that our upper bound for the true Jaccard similarity of the two sets is* $4/6 = 2/3$. *Now suppose that our predefined similarity threshold is set to* $S = 0.6$. *What this means is*

that we would treat the pair of sets (A, B) as such candidates which have a non-zero chance of being at least 0.6 *similar.*

As a consequence, we would perform the actual calculation of their true Jaccard similarity and get to the answer of

$$s(A,B) = \frac{|\{1,2\}|}{|\{1,2,3,4,5,6,7,8\}|} = 2/8 = 0.25.$$

As we can see, the pair of sets (A, B) constitutes a false positive pair with $S = 0.6$, as they seem to be an appealing pair for calculating their similarity, but in the end they turned out to have a much smaller similarity compared to what we were initially expecting.

Notice that should the similarity threshold have been $S = 0.7$ (instead of $S = 0.6$), the set of pairs (A, B) would no longer needed to be investigated for their true similarity, simply because $2/3 < 0.7$, implying that this pair of sets has zero probability of having a similarity exceeding 0.7.

5.2.2 *Bloom filters*

Bloom filters are **probabilistic data structure**s of great practical utility. This data structure can be used in place of standard set implementations (e.g. **hash set**s) in situations when the amount objects to store is possibly so enormous that it would be prohibitive to store all the objects in main memory. Obviously, by not storing all the objects explicitly, we pay some price as we will see, since we are not going to be able to tell with absolute certainty which object have been added to our data structure so far. However, when using bloom filters, we can exactly tell if an element *has not yet been added* to our data structure so far. Furthermore, given certain information about a bloom filter, we can also quantify the probability of erroneously claiming that a certain element has been added to it, when in reality it is not the case.

5.2.3 *The birthday party analogy*

One can imagine bloom filters according to the following birthday party analogy. Suppose we invite an enormous amount of people to celebrate our birthday. The number of our invitees can be so large that we might not even be able to enumerate them (just think of invitees indexed by the set of natural numbers). Since we would like to be a nice host, we ask every invitee in advance to tell us their favorite beverage, so that we can serve them accordingly. Unfortunately, purchasing the most beloved welcome drink for all our *potential* guest is out of practical utility for at least two reasons.

First of all, we invited a potentially infinitely many people. This means that – in theory – it is possible that we might need to purchase

infinitely many distinct beverages, which sounds like a very expensive thing to do. Additionally, it could be the case that some of our invitees will not eventually show up at the party in the end.

We realize that with a list of invitees so enormously large, our original plan of fulfilling the drinking preferences of all of our potential invitees is hopeless. Instead, what we do is that we create a shortlist of the most popular beverages, such as *milk, soda, champagne, vodka, apple juice, etc*, and ask them to tell us the kind of drink from our list they would be most happy to drink if they attended our party. We can look at the responses of our invitees as a **surjective function** of their preferences $p : \mathcal{I} \to \mathcal{B}$, i.e., a mapping which assigns every element from the set of invitees $\mathcal{I}$ to an element of the set of beverages $\mathcal{B}$. What surjectivity means in this context is that a single individual would always choose the very same welcome drink, but the same welcome drink might be preferred by multiple individuals.

For notational convenience, let m and n denote the number of invitees and the different kinds of beverages they can choose from. Furthermore, we can reasonable assume that $m \gg n$ holds. Notice that in the typical scenario, i.e., when the number of people in the party is far more than the number of beverages they are served, according to the **pigeonhole principle**, there must be certain beverages which must be consumed by more than a single guest in the party.

Now imagine that we would like to know who were those invitees who eventually made it to the party. We would like to know this answer, however, without being intrusive towards our guests and register them one-by-one, so we figure out the following strategy. Since everyone informed us in advance about the welcome drink they would consume upon their arrival at the party, we can simply look for those beverages that remained untouched by the end of the party from which fact we will be able to tell with absolute certainty that nobody who previously claimed to drink any of the beverages which remained sealed actually was there at our party.

? Are we going to be able the identify all the no-show invitees to the party by the strategy of looking for untouched bottles of beverages?

Talking in more formal terms, if a certain beverage b was not tasted by anyone in the party then we can be absolute certain that no individual i for which $p(i) = b$ was there at the party. Equivalently, for a particular invitee i, we can conclude with absolute certainty that i was not at the party if beverage $p(i)$ remained unopened during the entire course of the party.

On the other hand, the mere fact that some amount of beverage $p(i)$ is missing does not imply that invitee i was necessarily attending the party. Indeed, it can easily be the case that some other invitee i' just happens to prefer the same beverage as invitee i does, i.e., $p(i) = p(i')$, and it was invitee i' who is responsible for the missing amounts for beverage $p(i)$. Recall that such situations are inevitable

as long as $m < n$ holds. The behavior of bloom filters is highly analogous to the previous example.

5.2.4 *Formal analysis of bloom filters*

The previous analogy illustrated the working mechanisms and the main strengths and weaknesses of using a bloom filter. To recap, we have a way to map a possibly infinite set of objects of cardinality n to a much smaller set of finite values of cardinality m.

We would like to register which elements of our potentially infinite set we have seen over a period of time. Since it would be computationally impractical to store all of the objects that we have came across so far, we only store their corresponding hash value provided by a surjective function p. That way we lose the possibility to give an undoubtedly correct answer saying that we have seen a particular object, however, we can provide an unquestionably correct answer whenever we have not encountered a particular object so far.

This means that we are prone to **false positive error**s, i.e., answering yes when the correct answer would have been a no. On the other hand, we can avoid any **false negative error**s, i.e., using a bloom filter, we would never say it erroneously that an object has never been encountered before when it has been in reality.

The question that we investigate next is how to determine the amount of false positive rate for our bloom filter that we are using. Going back to the previous birthday party example, we can assume that function p distributes the set of individuals $\mathcal{I}$ evenly, i.e., roughly the same proportion of individuals are assigned to any of the m different beverages.

Notice that the m possible outcomes of function p defines m equivalence classes over the individuals. It follows from this assumption that all of the equivalence classes can be assumed to be of the same cardinality. In our particular case, it means that beverages can be expected to be equally popular, that is, the probability for a random individual to prefer any of the m beverages can be safely regarded as being equal to $\frac{1}{m}$.

Asking for the false positive rate of a bloom filter after storing a certain amount of objects in it, is equivalent to asking ourselves in the birthday party analogy the following question: given that we have a certain amount of distinct beverages and knowing that a certain amount of guests were present at the party in the end, what fraction of the beverages was consumed by at least one guest? Had a guest tasted a certain kind of beverage, we are no longer able to tell who the exact person was, hence such cases are responsible for the false positives.

In the followings, let m denote the distinct beverages we serve at the party as before, and introduce the variable $G < n$ for the number of guests – being a subset of the invitees – who eventually made it to our party. As we argued before, the probability of a drink to be tasted by a particular guest uniformly equals $\frac{1}{m}$. We are, however, interested in the proportion of those drinks which were consumed at least once over G independent trials.

The set of drinks that were consumed at least once can be equivalently referenced as the complement of the set of those drinks that were not tasted by anyone. Hence the probability of a drink being consumed can be expressed as the complement of the probability of a drink *not* being consumed by anyone. Hence, we shall work out this probability first.

The probability of a beverage not being consumed by a particular person is $1 - \frac{1}{m}$. The probability that a certain beverage is not consumed by any of the G guests, i.e., over G consecutive independent trials, can thus be expressed as

$$\left(1 - \frac{1}{m}\right)^G \approx e^{-G/m}, \tag{5.8}$$

with e denoting Euler constant. In order to see why the approximation in (5.8) holds, see the refresher below about Euler constant and some of its important properties.

So we can conclude that the probability of a particular beverage not being consumed over G consecutive trials is approximately $e^{-G/m}$. However, as we noted before, we are ultimately interested in the probability of a beverage being drunk at least once, which is exactly the complement of the previous probability. From this, we can conclude that the false positive rate of a bloom filter with m buckets and G objects inserted can be approximated as

$$1 - e^{-G/m}. \tag{5.9}$$

Notice that in order to keep the false positive errors low, we need to try to keep $\frac{G}{m}$ as close to zero as possible. There are two ways we can achieve this, i.e., we might try to keep the value of G low or we could try to increase m as much as possible. The first option, that is trying to persuade invitees not to become actual guests and visit the party, does not sound to be a viable way to go.

However, we could try to increase the number of beverages to the extent our budget allows us. In terms of bloom filters, this means that we should try to partition our objects into as many groups as possible, i.e., we should strive for the allocation of a bitmask which has as many cells we can afford.

Basically, the false positive rate of the bloom filter is affected by the average number of objects belonging to a certain hash value em-

ployed, i.e., $\frac{G}{m}$. As this fraction gets higher, we should be prepared to see an increased frequency of false positive alarms.

MATH REVIEW | EULER'S CONSTANT

Euler's coefficient (often denoted by e) can be defined as a limit, i.e.,

$$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n \approx 2.7182818.$$

Figure 5.15 displays how fast this above sequence converges to the value of e.

Relying on that limit, we can give good approximations of expressions of the form

$$\left(1+\frac{1}{n}\right)^n$$

for large values of n, i.e., it tends towards the value of the Euler constant e. Analogously, expressions of the form

$$\left(1+\frac{1}{n}\right)^k$$

can be rewritten as

$$\left[\left(1+\frac{1}{n}\right)^n\right]^{\frac{k}{n}},$$

which can be reliably approximated by

$$e^{\frac{k}{n}},$$

when the value of n is large. The approximation also holds up to a small modification, when a subtraction is involved instead of an addition, i.e.,

$$\left[\left(1+\frac{1}{n}\right)^n\right]^{\frac{k}{n}} \approx \frac{1}{\exp^{\frac{k}{n}}} = e^{-\frac{k}{n}}.$$

Figure 5.14: Euler's constant

Example 5.8. *We can easily approximate the value of* 0.99^{100} *with the help of Euler's coefficient if we notice that* $0.99 = 1 - \frac{1}{100}$, *hence* $0.99^{100} = \left(1-\frac{1}{100}\right)^{100}$. *As* $n = 100$ *can be regarded as a large enough number to give a reliable approximation of the expression in terms of Euler's coefficient, we can conclude that* $0.99^{100} \approx e^{-1} \approx 0.36788$. *Calculating the true value for* 0.99^{100} *up to 5 decimal points, we get* 0.36603, *which is pretty close to the approximated value of* e^{-1}.

Example 5.9. *As another example, we can also approximate the value of* 1.002^{200} *relying on Euler's coefficient. In order to do so, we have to note that* $1.002 = 1 + \frac{1}{500}$, *hence* $1.002^{200} = \left(\left(1+\frac{1}{500}\right)^{500}\right)^{\frac{200}{500}}$. *This means*

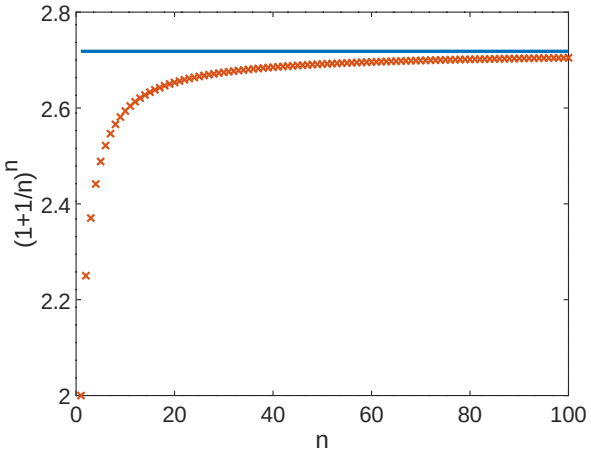


Figure 5.15: Illustration of the Euler coefficient as a limit of the sequence $(1+\frac{1}{n})^n$.

that 1.002^{200} *can be approximated as* $\sqrt[5]{e^2} \approx 1.49182$. *Indeed, the true value of* $1.002^{200} = 1.49122$ *(up to 5 decimal points) is again very close to our approximation.*

5.2.5 *Bloom filters with multiple hash functions*

We can extend the previous birthday party analogy in a way that every guest is served k welcome drinks. Invitees enjoy freedom in setting up their compilation of welcome drinks, meaning they can choose their k beverages the way they wish. For instance, there is no restriction on the welcome drinks to be distinct, so in theory it is possible that someone drinks k units of the very same drink. In terms of a bloom filter what it means is that we employ multiple independent surjective hash functions to the objects that we wish to store.

Now that everyone drinks k units of beverages, our analysis also requires some extensions. When we check whether a certain person visited our party, we shall check *all* k *beverages* he/she applied for duing our query and if we see *any of those* k *beverages* to be untouched by anyone, we can certainly come to the conclusion that the particular person did not show up at the party. Simultaneously, a person might be *falsely conjectured* to have visited the party, *if all the beverages* he/she declared to drink got opened and consumed by *someone*.

It seems that we can decrease the false positive rate when using the extended version of bloom filters with k *independent* hash functions, since we would arrive at erroneous positive answers only if for a given object, k independent hash functions map it to such buckets that have been already occupied by some other object at least once. What this suggests is that we should include an exponentiation in Eq. (5.9) and obtain $\left(1-e^{-G/m}\right)^k$.

? Can you think of a reason why applying multiple hash functions can increase the probability of false positive hits?

A bit of extra thinking, however, brings us to the observation that the above way for calculating the false positive rate is too optimistic as it does not account for the fact that objects are responsible for modifying the status of more than just one bucket of the bloom filter. In fact every object has the potential of modifying up to k distinct buckets.

Recall that the average load factor (G/m) is an important component in the analysis of the false positive rate of bloom filters. In the extended version of bloom filters with k independent hash functions, we can approximate the load factor by $\frac{Gk}{m}$, hence the final false positive rate becomes

$$\left(1 - e^{-\frac{Gk}{m}}\right)^k. \tag{5.10}$$

As argued previously, we can reduce the false positive rate of a bloom filter by increasing k, since Eq. (5.10) includes raising a probability to the k^{th} power, and $\lim_{k\to\infty} p^k = 0$, whenever $p < 1$. Looking at Eq. (5.10), however, also suggests that the false positive rate can be affected negatively by increasing the value for k. To see why this is the case, notice that the false positive rate implicitly depends on the probability of a bucket not being used, i.e., $e^{-\frac{Gk}{m}}$. The larger the previous value is, or equivalently, the more fraction $\frac{Gk}{m}$ tends to zero, the better we can reduce the false positive rate. Increasing k in the nominator certainly acts as a counterforce for the previous fraction for going towards zero.

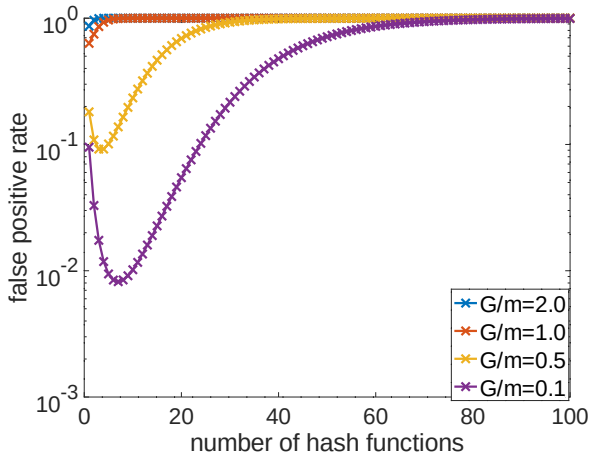


Figure 5.16: False positive rates of a bloom filter with different load factors as a function of the hash functions employed per inserted instances. The different load factors corresponds to different average number of objects per a bucket (G/m).

Figure 5.16 illustrates this dual role of the number of hash functions applied in a bloom filter, k. Even though a higher value of k allows us to check membership in a bloom filter based on a more complex criterion which could reduce false positive rate, it also makes the average load factor of the bloom filter higher, increasing the chance

for a false positive answer on the other hand. Inspecting Figure 5.16 reveals us that irrespectful of the initial load factor of the bloom filter (G/m), increasing the number of hash functions employed beyond 5 does not seem to provide any benefit for reducing the false positive rate.

5.3 *Summary of the chapter*

This chapter introduced approximate techniques for identifying similar objects in potentially large datasets. One way for doing that is the locality sensitive hashing (LSH) technique using which we can potentially fail to detect certain pairs of objects with in reality has a high degree of similarity with each other. Another form of error is when we needlessly perform pairwise comparisons between pairs of points that are highly dissimilar to each other otherwise. The former type of error is that of false positives and the latter form is that of false negatives. Throughout the chapter we investigated LSH from a theoretical point of view and quantified its error rate. At the end of the chapter, we introduced algorithms and a data structure that are solely affected by false positive errors.

Johnson et al. [2017] introduces a highly efficient software package[1] which is a recommended source for readers who are interested in a highly efficient (and hardware accelerated) implementation of the techniques discussed in this chapter.

[1] accessible from https://github.com/facebookresearch/faiss

6 | Dimensionality reduction

Learning Objectives:
- The curse of dimensionality
- Principal Component Analysis
- Singular Value Decomposition
- CUR decomposition
- Linear Discriminant Analysis

This chapter introduces dimensionality reduction techniques. Readers finishing the chapter are expected to

- develop an intuition how dimensionality reduction can help data mining applications,
- be able to explain the differences between different various dimensionality reduction techniques,
- quantitatively assess the quality of the dimensionality reduction algorithms,
- understand and argue for the need of a certain kind of dimensionality reduction technique for a given dataset.

Data mining applications typically deal with high dimensional data. From a human mindset, even a space in 100 dimensions is incredibly hard to imagine, yet it is not uncommon that data mining applications need to efficiently deal with hundreds of thousands or even millions of dimensions at a time.

As such each data instance can be conventionally imagined as a vector in some high-dimensional space, i.e., samples are described by a collection of observations which can be naturally encoded as a series of scalars. Section 3.2.1 already discussed how nominal features are typically turned into numerical features, so we can assume without loss of generality here, that every data point can be described relying on a series of scalars in the form of a vector.

Decreasing the original dimensionality of our data points can be motivated in multiple ways. Probably the most profound is that after decreasing the dimensionality of datasets, we can store them on a reduced amount of space. Besides this obvious benefit, it is often the case that by transforming some data into a lower dimensional space it is possible to get rid of some of the irrelevant noise that is captured in the high dimensional representation of the data. As such, dealing with such a variant of some dataset the dimensionality of which is

reduced, we can not only process it mode effectively, but often obtain better results when doing so.

? Think of a few datasets for which data points are described by vectors of radically different dimensions.

6.1 *The curse of dimensionality*

It is important to notice that from certain aspects high dimensional spaces behave counter-intuitively. We can easily convince ourselves about the strange behavior of high dimensional spaces if we investigate **hypersphere**s in d dimensions as d tends to infinity.

Hyperspheres are nothing else but the generalization of circles beyond 2 dimensions. That is such a collection of points for which the squared sum of coordinates sum up to some pre-defined value r^2, with r denoting the radius of the hypersphere. This definition is in line with the regular way we define a circle, i.e., the collection of 2D points $\boldsymbol{x} = [x_1 x_2]$ for which $x_1^2 + x_2^2 = r^2$.

The volume of a hypershpere in d dimensions and radius r is given by the formula

$$V_d(r) = \frac{\pi^{d/2}}{\Gamma\left(\frac{d}{2}+1\right)} r^d, \tag{6.1}$$

with Γ denoting the Γ function shortly summarized in Figure 6.2.

Recall that Eq. (6.1) is simply the general form of the two dimensional case, i.e., the formula boils down to πr^2 when $d = 2$, since $\Gamma\left(\frac{d}{2}+1\right)$ is just $\Gamma(2) = 1! = 1$ in that case.

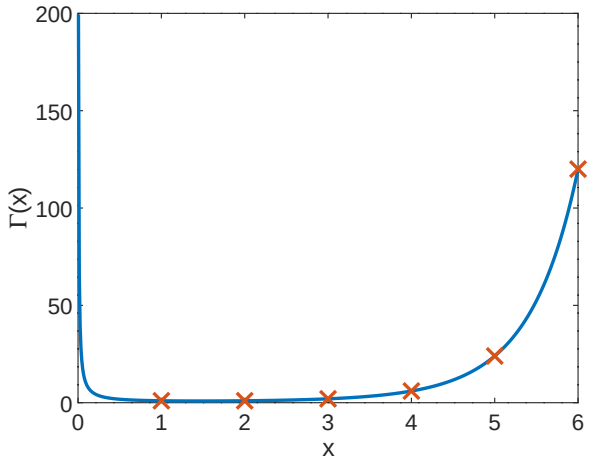


Figure 6.1: Gamma function over the interval [0.01,6] with integer values denoted with orange crosses.

Unit hyperspheres are just hyperspheres with unit long radius. The first strange thing we can notice regarding the volume of such hyperspheres is that it goes to zero as the dimension of the space increases. We can see this by looking at Eq. (6.2) and see that the factorial-like part in the denominator grows strictly quicker than the

MATH REVIEW | GAMMA FUNCTION

The **gamma function** is formally defined as

$$\Gamma(x) = \int_0^\infty t^{s-1} e^{-t} dt.$$

This function can be viewed as a generalization of the factorial function beyond non-integer values as well. Indeed, the identity $\Gamma(i) = (i-1)!$ holds for every integer $i > 0$. As such, the gamma function has a similar growth rate as the factorial. Its fast increase in the function value (indicated by blue line) even for moderate inputs and its connection to the values of the factorial function (indicated by red crosses) is depicted in Figure 6.1.

Figure 6.2: Gamma function

exponential part in its nominator. Indeed, the volume of the unit hypersphere is depicted in Figure 6.3 illustrating that the volume of unit hyperspheres grow up to five dimensions, but drops quickly beyond that point.

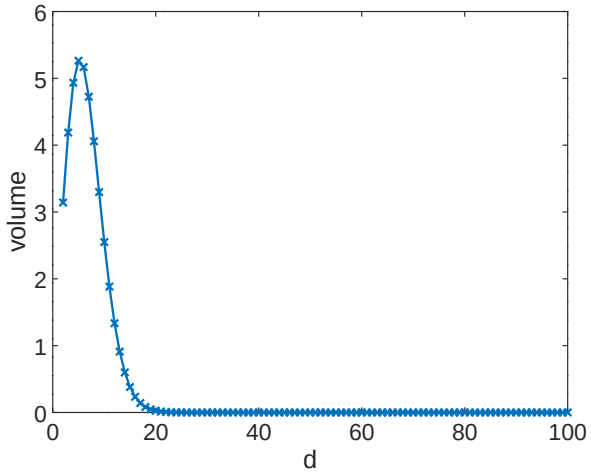


Figure 6.3: The volume of a unit hypersphere as a function of the dimensionality d.

It is then instructive to compare the volumes of two hyperspheres in d dimensions (called **d-spheres** in short) with radii 1 and $(1-\epsilon)$, i.e., the fraction

$$\frac{V_d(1) - V_d(1-\epsilon)}{V_d(1)}. \tag{6.2}$$

This quantity gives us the portion of the volume of the d-sphere with radius 1 lying beyond radius $1-\epsilon$, that is, close to the surface and far from the origin. Since the first fractional part in Eq. (6.1) can be treated as a constant which only scales the result and cancels out anyway in Eq. (6.2), hence the ratio can be equivalently written as

$1 - (1 - \epsilon)^d$.

As a consequence, high-dimensional hyperspheres have the intriguing property that the majority of their mass is concentrated around their surface as opposed to being evenly distributed. The fraction of volume for a unit hypersphere which is in the ϵ neighborhood of its surface is visualized in Figure 6.4 as a function of the dimensionality of the space. The light blue cross for the $\epsilon = 0.1$ curve indicates that nearly 90% of the volume of a 20-dimensional unit sphere has distance at least $1 - \epsilon = 0.9$ units from the origin.

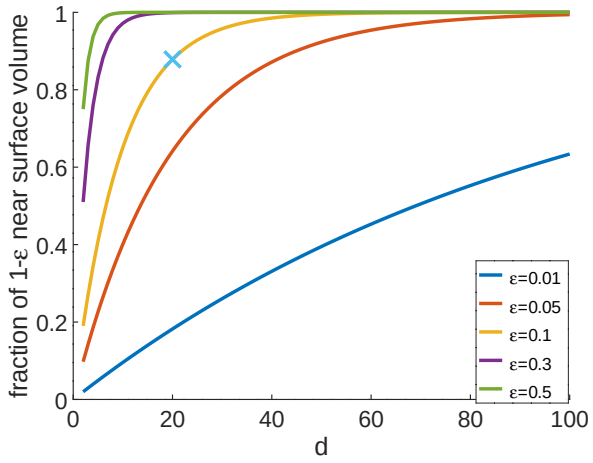
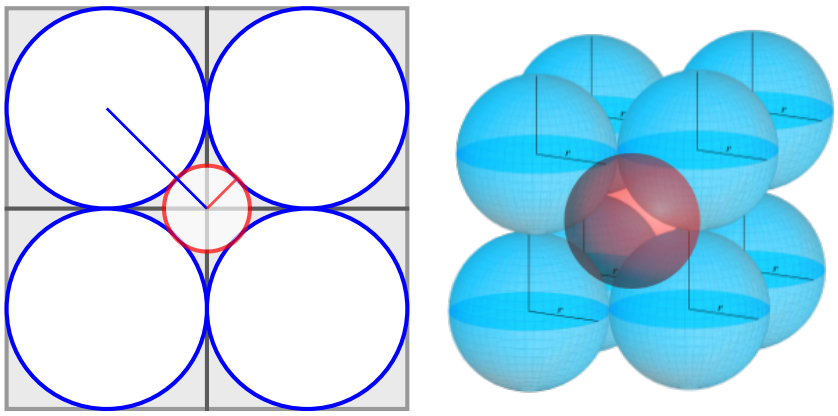


Figure 6.4: The relative volume of the $1 - \epsilon$ sphere to the unit sphere.

Now imagine a hypercube – a square in 2-dimensions and a cube in 3-dimensions – which has sides of length of 4. Such a hypercube can accommodate 2^d non-intersecting hyperspheres with unit radius as illustrated in Figure 6.5 by the blue objects. Now imagine, that we want to squeeze in an additional origin-centered hypersphere which touches all of the unit hyperspheres.



(a) 2-dimensional case

(b) 3-dimensional case

Figure 6.5: The position of a (red) hypersphere which touches the (blue) hyperspheres with radius 1.

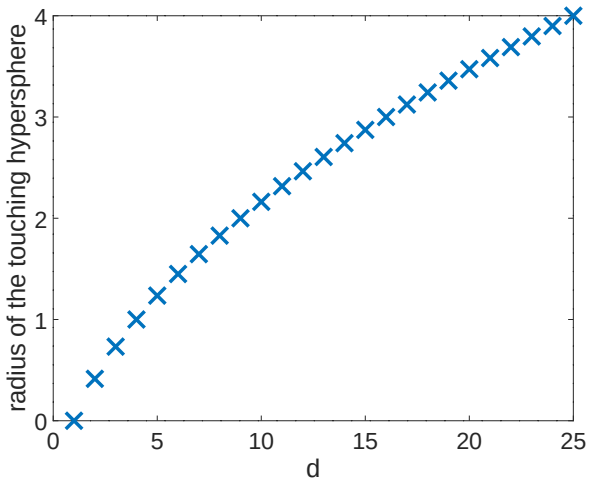


Figure 6.6: The radius of the hypersphere in d dimensions which touches the 2^d non-intersecting unit hyperspheres located in a hypercube with sides of length of 4.

One can see it from Figure 6.5 (a) that the line segment going from the origin to any of the centers of the four unit 2-spheres have length of $\sqrt{2}$. One of such segments is included in Figure 6.5 (a) with blue color. As a consequence of the previous observation, the radius of the hypersphere which touches the unit hyperspheres needs to be $\sqrt{2} - 1$.

This observation generalizes inductively to higher dimensions, i.e., the radius of the d-sphere which touches all the other 2^d non-intersecting unit radius d-spheres will have a radius of $\sqrt{d} - 1$ as illustrated in Figure 6.6. What this implies is that the touching hypersphere would eventually step out of the hypercube with 4 units of length accommodating the unit hyperspheres once we surpass 9 dimensions.

We discuss a final artifact of high-dimensional spaces, namely, that pairwise distances between data points become less meaningful as the dimensionality of the space increases. This is because distances between pairs of points tend to become very concentrated without a real variance, that is most of the pairwise distances are end up having the same (large) value. This impose a problem which is often referred as the **curse of dimensionality**. Indeed, if pairwise distances between points are very similar to each other then talking out nearest (least distant) pairs of observations become more nebulous.

Figure 6.7 contains the distribution of the pairwise distances between uniformly sampled points of substantially different dimensions, $d \in \{2, 10, 100, 1000\}$. Figure 6.7 visually supports the previously mentioned property, i.e., as d grows the distribution of the pairwise distances gets more concentrated and its mean value also increases.

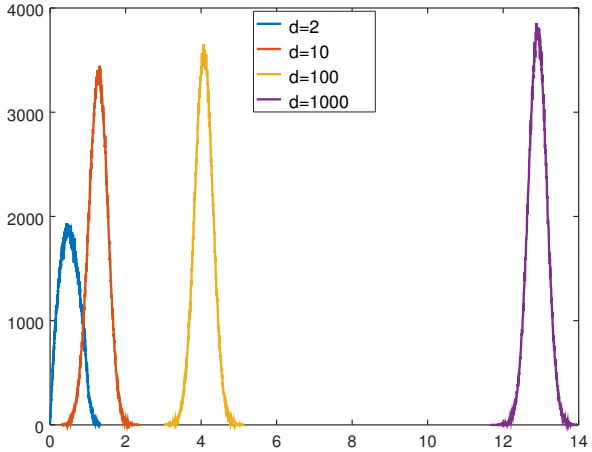


Figure 6.7: The distribution of the pairwise distances between 1000 pairs of points in different dimensional spaces.

6.2 *Principal Component Analysis*

The key idea behind principal component analysis (and other dimensionality reduction techniques) is that even though we often face high dimensional observations, these data points typically lay on a relatively low dimensional manifold. For instance, consider the exemplar user-item dataset included in Table 6.1, which illustrates how many times a certain person read a particular newspaper during the course of one week. We can see that even though the people are initially represented as 3-dimensional objects (reflecting the number of times they read a certain newspaper along the different axes), we can still accurately represent them in two dimensions with a different choice of axes for our coordinate system. In this simplified example, we can easily define the optimal axes for the new representation if we observe the linear dependence between the columns of the sample matrix in Table 6.1.

	Newspaper		
	A	B	C
Evelyn	1	1	0
Mark	3	3	0
Peter	0	0	4
Martha	0	0	2

Table 6.1: Example user-item dataset with a lower than observed effective dimensionality.

Upon dimensionality reduction, we are mapping the original observations $\boldsymbol{x}_i$ into some "compressed" representation that we denote by $\tilde{\boldsymbol{x}}_i$. Ultimately, we are looking for such a linear mapping in case of **principal component analysis** (**PCA**) which minimizes the distortion between the original data representation and its compressed

counterpart. This means that our objective can be expressed as

$$\sum_{i=1}^{n} \|(\boldsymbol{x}_i - \tilde{\boldsymbol{x}}_i)\|_2^2. \tag{6.3}$$

Notice that the above objective value can be equivalently rewritten as

$$\sum_{i=1}^{n} \|(\boldsymbol{x}_i - \boldsymbol{\mu}) - (\tilde{\boldsymbol{x}}_i - \boldsymbol{\mu})\|_2^2, \tag{6.4}$$

where $\boldsymbol{\mu}$ simply denotes the average observation in our dataset. Subtracting the mean observation from every data point in the objective function expressed in Eq. (6.3) simply corresponds to mean centering the dataset as already discussed in Section 3.2.1.

First, let us try to come up with an approximation of the observed data with zero degree of freedom, that is try to approximate all observations $\boldsymbol{x}_i$ by a 'static' $\tilde{\boldsymbol{x}}$ being insensitive to the particular observation we are assigning it as its image. What this practically means is that we can rewrite and simplify Eq. (6.4) as

$$\sum_{i=1}^{n} (\boldsymbol{x}_i - \boldsymbol{\mu})^\intercal (\boldsymbol{x}_i - \boldsymbol{\mu}) - 2(\tilde{\boldsymbol{x}} - \boldsymbol{\mu})^\intercal \sum_{i=1}^{n} (\boldsymbol{x}_i - \boldsymbol{\mu}) + \sum_{i=1}^{n} (\tilde{\boldsymbol{x}} - \boldsymbol{\mu})^\intercal (\tilde{\boldsymbol{x}} - \boldsymbol{\mu}). \tag{6.5}$$

At first sight, Eq. (6.5) might not seem as a simplified version of Eq. (6.4), however, we can take two crucial observations about Eq. (6.5), i.e.,

- the first summation does not include $\tilde{\boldsymbol{x}}$, hence it can be treated as a constant which does not affects the optimum for $\tilde{\boldsymbol{x}}$, meaning that the entire sum can be dropped without modifying the optimum,
- the summation in the second term evaluates to the zero vector by the definition of $\boldsymbol{\mu}$, i.e., $\sum_{i=1}^{n} (\boldsymbol{x}_i - \boldsymbol{\mu}) = \mathbf{0}$. As the dot product of any vector with the zero vector is always going to be zero, the second term can also be dropped.

What we can conclude based on the previous observations is that minimizing the original objective for a 'static' $\tilde{\boldsymbol{x}}$ is equivalent to finding such a value for $\tilde{\boldsymbol{x}}$ which minimizes

$$\sum_{i=1}^{n} (\tilde{\boldsymbol{x}} - \boldsymbol{\mu})^\intercal (\tilde{\boldsymbol{x}} - \boldsymbol{\mu}) = n\|\tilde{\boldsymbol{x}} - \boldsymbol{\mu}\|_2^2, \tag{6.6}$$

which can be trivially minimized by the choice of $\tilde{\boldsymbol{x}} = \boldsymbol{\mu}$. What this result tells us is that if we want to represent all our observations by a single vector, this vector should be chosen as the average of our data points, because this one can minimize the overall sum of squared distances to the original data points.

Based on the previous result, let us make a more complex approximation to our data points in the following form: $\tilde{\boldsymbol{x}}_i = a_i\boldsymbol{e} + \boldsymbol{\mu}$. That is, instead of simply representing every observation by $\boldsymbol{\mu}$, we also introduce an additional offset term $a_i\boldsymbol{e}$ for each data point. These offsets are such that all points rely on the same offset direction defined by $\boldsymbol{e}$, whereas they have their individual magnitude parameter a_i as well. What this means is that our objective function this time changes to

$$\sum_{i=1}^{n} \|\boldsymbol{x}_i - (a_i\boldsymbol{e} + \boldsymbol{\mu})\|_2^2 = \sum_{i=1}^{n} \|(\boldsymbol{x}_i - \boldsymbol{\mu}) - a_i\boldsymbol{e}\|_2^2, \tag{6.7}$$

which expression can be brought to the equivalent form of

$$\sum_{i=1}^{n} (\boldsymbol{x}_i - \boldsymbol{\mu})^\intercal(\boldsymbol{x}_i - \boldsymbol{\mu}) - 2\sum_{i=1}^{n} a_i\boldsymbol{e}^\intercal(\boldsymbol{x}_i - \boldsymbol{\mu}) + \sum_{i=1}^{n} a_i\boldsymbol{e}^\intercal\boldsymbol{e}a_i. \tag{6.8}$$

Notice that at this point, upon the minimization of Eq. (6.8), we definitely have infinitely many solutions as the very same $a_i\boldsymbol{e}$ offsets (including the optimal one) can be expressed in infinitely many ways. In order to see why this is the case, just imagine a particular offset, given by $a_i\boldsymbol{e}$. We can now take any non-zero vector $\boldsymbol{e}'$ pointing to the same direction of $\boldsymbol{e}$, say $\boldsymbol{e}' = 2\boldsymbol{e}$, and define our original offset $a_i\boldsymbol{e}$ simply as $\frac{1}{2}a_i\boldsymbol{e}' = (\frac{1}{2}a_i)(2\boldsymbol{e}) = a_i\boldsymbol{e}$. Since we can find an appropriate coefficient for every non-zero $\boldsymbol{e}'$ pointing to the same direction as $\boldsymbol{e}$, this tells us that up to this point there are indeed infinite solutions to our problem. To avoid this ambiguity in the solution we search for, we will require it to be of some predefined length, in particular we will only be accepting solutions for which $\|\boldsymbol{e}\| = 1$, or equivalently, $\boldsymbol{e}^\intercal\boldsymbol{e} = 1$, holds. This kind of optimization, i.e. when we make certain restrictions on the acceptable solutions is called constrained optimization that we will elucidate via a simple example now briefly.

Let us see now through a concrete example how constrained optimization can be efficiently solved in practice with the help of Lagrange multipliers. The kind of optimization that we solve for obtaining the optimal solution for PCA is similar in vein to the following simple example. Readers, who want to read a more detailed introductory tutorial on Lagrange multipliers are encouraged to read Klien [2004][1].

[1] Dan Klien. Lagrange Multipliers Without Permanent Scarring. August 2004. URL www.cs.berkeley.edu/~klein/papers/lagrange-multipliers.pdf

Example 6.1. *Let us try to minimize the function $f(x, y) = 12x + 9y$ given the additional constraint that the minimizer of the function has to be located on the unit circle. That is, the only feasible solutions are such points for which $x^2 + y^2 = 1$ is met.*

It can be noted that without any restrictions on the values of x and y, the function $f(x, y)$ can be made arbitrarily small, since once x and y tends to negative infinity the function value also decreases without any bound towards negative infinity.

MATH REVIEW | CONSTRAINED OPTIMIZATION

In the case of **constrained optimization** problems, we are not simply about to optimize for a certain function $f(x)$, but simultaneously, we want to do it so, such that we respect certain constraining conditions – defined by additional functions $g_i(x)$ – towards the optimal solution $\boldsymbol{x}$. (Non-)linear **constrained optimization** problems are thus of the following form

$$\begin{aligned} & f(\boldsymbol{x}) \rightarrow min/max \\ \text{such that } & g_i(\boldsymbol{x}) = 0 \qquad \forall i \in \{1, \ldots, n\} \end{aligned}$$

Lagrange multipliers offer a schema for solving such constrained optimization problems. The solution first goes by defining the **Lagrange function** as

$$L(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) - \sum_{i=1}^{n} \lambda_i g_i(\boldsymbol{x})$$

By applying the **Karush-Kuhn-Tucker (KKT) conditions** we can obtain a possible solution for our constrained optimization problem which does not violate the constraints involved in the particular optimization problem we are solving for.

$$\nabla L(\boldsymbol{x}, \boldsymbol{\lambda}) = 0 \tag{6.9}$$

$$\lambda_i g_i(\boldsymbol{x}) = 0 \forall i \in \{1, \ldots, n\} \tag{6.10}$$

$$\lambda_i \geq 0 \tag{6.11}$$

Note that the KKT conditions only provide a necessity condition for finding an optimal $\boldsymbol{x}$, i.e., they might as well find such an $\boldsymbol{x}$ which obey for the conditions and provides a saddle point for $f(x)$ and not a genuine minimizer/maximizer for it.

Figure 6.8: Constrained optimization

We can find a solution for the above constrained optimization problem using Lagrange multipliers by treating

$$f(x, y) = 12x + 9y$$

and

$$g(x, y) = x^2 + y^2 - 1.$$

The Lagrange function that we construct from these functions is

$$\begin{aligned} L(x, y, \lambda) &= 12x + 9y + \lambda(x^2 + y^2 - 1) \\ &= 12x + 9y + \lambda x^2 + \lambda y^2 - \lambda \end{aligned}$$

In order to find a potential optimum for the function f respecting the constraints given by g, we need to find such values of x, y, λ for which the gradient of the Lagrangian equals the zero vector $\mathbf{0}$.

As a reminder, the gradient of a function is simply the vector obtained by taking the partial derivatives of the function with respect its variables. This means that we are looking for a solution, where

$$\nabla L(x, y, \lambda) = \begin{bmatrix} \frac{\partial L(x,y,\lambda)}{\partial x} \\ \frac{\partial L(x,y,\lambda)}{\partial y} \\ \frac{\partial L(x,y,\lambda)}{\partial \lambda} \end{bmatrix} = \begin{bmatrix} 12 + 2\lambda x \\ 9 + 2\lambda y \\ x^2 + y^2 - 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{0}.$$

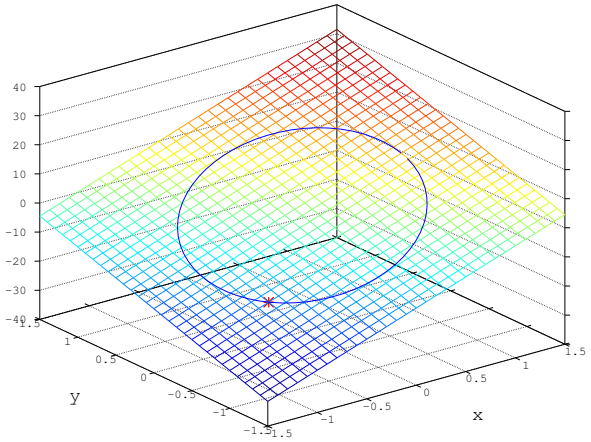


Figure 6.9: An illustration of the solution of the constrained minimization problem for $f(x, y) = 12x + 9y$ with the constraint that the solution has to lie on the unit circle.

This gives us a system of equations and rearranging its first two rows yields that $x = -\frac{6}{\lambda}$ and $y = -\frac{4.5}{\lambda}$. Plugging these values into the third equation and solving for λ gives us the result $\lambda = 7.5$. Substituting this value into the previously determined equations, we get $x = -\frac{6}{7.5} = -0.8$ and $y = -\frac{4.5}{7.5} = -0.6$.

We can verify that the point $\begin{bmatrix} -0.8 \\ -0.6 \end{bmatrix}$ indeed lies on the unit circle, as $(-0.8)^2 + (-0.6)^2 = 0.64 + 0.36 = 1$. Additionally, as Figure 6.9 illustrates it by the red star marker, this point is indeed, a minimizer of the function in question with the given constraint.

After the short recap on constrained optimization and Lagrange multipliers, we can turn back to our discussion on PCA. Looking back at Eq. (6.8), we can make a similar observation as we did for Eq. (6.5), namely that the first sum is independent from the parameters that we are trying to set optimally, hence omitting it from our objective does not influence our optimal solution. Unfortunately, unlike in the case of Eq. (6.5), the second term of Eq. (6.8) will not evaluate to zero necessarily, so we cannot simply discard that part this time. On the other hand, we can notice that the last summation

in Eq. (6.8) can be expressed simply as $\sum_{i=1}^{n} a_i^2$ – due to the constraint that we introduced for $\boldsymbol{e}$, i.e., that $\boldsymbol{e}^\intercal \boldsymbol{e} = 1$ has to hold. Based on these, we can equivalently express our objective from Eq. (6.7) as

$$-2\sum_{i=1}^{n} a_i \boldsymbol{e}^\intercal (\boldsymbol{x}_i - \boldsymbol{\mu}) + \sum_{i=1}^{n} a_i^2. \tag{6.12}$$

Taking the partial derivative of Eq. (6.12) with respect a_i and setting it to zero, we get that for the optimal solution

$$a_i = \boldsymbol{e}^\intercal (\boldsymbol{x}_i - \boldsymbol{\mu}) \tag{6.13}$$

has to hold. Plugging in the optimal values that we have just determined for the a_i variables into Eq. (6.12) we get an equivalent expression as

$$-2\sum_{i=1}^{n} \boldsymbol{e}^\intercal (\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{x}_i - \boldsymbol{\mu})^\intercal \boldsymbol{e} + \sum_{i=1}^{n} \boldsymbol{e}^\intercal (\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{x}_i - \boldsymbol{\mu})^\intercal \boldsymbol{e}, \tag{6.14}$$

that we can rewrite as

$$-2\boldsymbol{e}^\intercal \left(\sum_{i=1}^{n} (\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{x}_i - \boldsymbol{\mu})^\intercal \right) \boldsymbol{e} + \boldsymbol{e}^\intercal \left(\sum_{i=1}^{n} (\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{x}_i - \boldsymbol{\mu})^\intercal \right) \boldsymbol{e}, \tag{6.15}$$

since we can move out vector $\boldsymbol{e}$ from the summations in Eq. (6.14). Now the minimization problem in Eq. (6.15) exactly equals to minimizing

$$-\boldsymbol{e}^\intercal \left(\sum_{i=1}^{n} (\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{x}_i - \boldsymbol{\mu})^\intercal \right) \boldsymbol{e},$$

or equivalently maximizing the previous expression without negating it, i.e., the solution we are looking for is such that

$$\boldsymbol{e}^\intercal \left(\sum_{i=1}^{n} (\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{x}_i - \boldsymbol{\mu})^\intercal \right) \boldsymbol{e} \tag{6.16}$$

gets maximized. We can notice that the expression within the parenthesis in Eq. (6.16) simply equals by definition the **scatter matrix** of the dataset which was already covered earlier in Figure 3.5 of Section 3.2.3.

After a series of observations, we can come to the conclusion that the minimization of Eq. (6.7) is essentially the same problem as the maximization of $\boldsymbol{e}^\intercal S \boldsymbol{e}$, for such $\boldsymbol{e}$ vectors the (squared) norm of which equals 1, and S denotes the scatter matrix derived from our observations. By relying on the method of Lagrange multipliers for solving constrained optimization problems, we get the necessity (and this

time also sufficiency) condition for such an optimum which respects the constraint on $\boldsymbol{e}$ being unit norm as

$$S\boldsymbol{e} = \lambda \boldsymbol{e}, \tag{6.17}$$

which result simply suggests that the optimal $\boldsymbol{e}$ must be one of the eigenvectors of the scatter matrix S.

Recall that an $n \times n$ matrix comes with n (eigenvalue, eigenvector) pairs. Now that we realized that the optimal $\boldsymbol{e}$ has to be an eigenvector of the scatter matrix, we also know that the expression that we are optimizing for will take on the value $\boldsymbol{e}^\intercal(\lambda \boldsymbol{e}) = \lambda(\boldsymbol{e}^\intercal \boldsymbol{e})$, which simply equals λ, simply because the optimal solution we are looking for fulfills $\boldsymbol{e}^\intercal \boldsymbol{e} = 1$. Since we are maximizing the quadratic expression $\boldsymbol{e}^\intercal S \boldsymbol{e}$, it means that we should choose that eigenvector of S which corresponds to the highest eigenvalue, i.e., the **principal eigenvector**. Since scatter matrices are symmetric and positive (semi)definite, we can be sure that the eigenvalues are non-negative real values, so that we can index them in a way that $\lambda_1 \geq \lambda_2 \geq \ldots \lambda_n > 0$ relations hold.

MATH REVIEW | EIGENVALUES OF SCATTER AND COVARIANCE MATRICES

Figure 3.5 already provided a refresher on the concept of **scatter matrices** and **covariance matrices**. At this point we repeat if briefly, that for a given dataset, the two matrices only differ from each other in a constant multiplicative factor, i.e., $C = \frac{1}{n} S$, with $C \in \mathbb{R}^{n \times n}$ denoting the covariance matrix of some dataset $X \in \mathbb{R}^{n \times m}$, and S referring to the scatter matrix.
Notice that both S and C can be viewed as a **Gramian matrix**, i.e., a matrix which can be expressed in the form of the product of some matrix and its transpose. Gramian matrices have a bunch of nice properties. Every Gramian matrix – being symmetric and positive semi-definite – always has real and non-negative eigenvalues.

Figure 6.10: Eigenvalues of scatter and covariance matrices

Example 6.2. *Suppose our data matrix is the following*

$$M = \begin{bmatrix} 5 & 4 & 0 \\ 4 & 5 & 0 \\ 1 & 1 & 4 \\ 2 & 5 & 5 \end{bmatrix}$$

In order to find its 1–dimensional representation according to PCA, we need to perform the following steps:

0. (optional, but good to have) preprocess the data (e.g., standardize)

1. determine its scatter matrix,

2. *find its dominant eigenvector* $\boldsymbol{x}$,

3. *project the observations onto* $\boldsymbol{x}$.

The Octave equivalent of the above steps for our example dataset in matrix M are summarized in Figure 6.11.

CODE SNIPPET

```
M=[5 4 0; 4 5 0; 1 1 4; 2 5 5];
C = cov(M);
[eig_vecs, eig_vals] = eig(C);
>>
eig_vecs =

-0.836080   0.054461  -0.545897
0.272514   0.904842  -0.327105
-0.476136   0.422250   0.771362

eig_vals =

Diagonal Matrix

0.21368     0           0
0           2.96484     0
0           0          10.65482

[dominant_eigenvalue, dominant_column]=max(diag(eig_vals));
dominant_eigenvector = eig_vecs(:, dominant_column);
compressed_data = M * dominant_eigenvector
compressed_variance = var(compressed_data)
>>
compressed_data =

-4.0379
-3.8191
2.2124
1.1295

compressed_variance = 10.655
```

Figure 6.11: Calculating the 1–dimensional PCA representation of the dataset stored in matrix M.

6.2.1 *An example application of PCA – Eigenfaces*

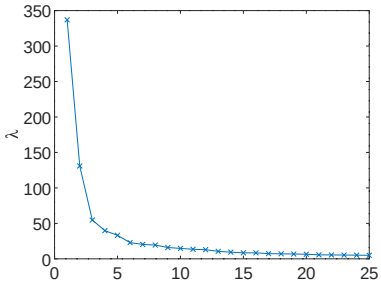
We demonstrate how principal component analysis works via a collection of 5,000 32×32 pixel gray-scale face images. A sample of 25 such images can be seen in Figure 6.12. The data matrix X that we work with contains gray scale pixel intensities and it has a shape of 5000×1024.

Figure 6.12: A 25 element sample from the 5,000 face images in the dataset.

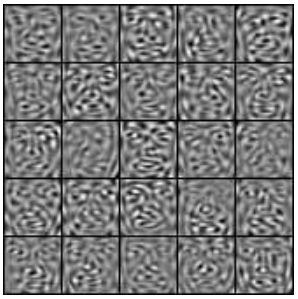
Since the above described dataset has a covariance matrix of shape 1024×1024, we can determine 1024 (eigenvalue, eigenvector) pairs. We can conveniently handle the 1024-dimensional eigenvectors as if they were $1024 = 32 \times 32$ gray scale images themselves. Figure 6.13 contains the visualization of eigenvectors with eigenvalues of different magnitudes. Figure 6.13 reveals that the eigenvectors determined during PCA can be viewed as *prototypical face images* that are used for the reconstruction of any image. The larger eigenvalue corresponds to an eigenvector of greater utility. From an intuitive point of view, it also verifies the strategy of projecting the data points to the eigenvectors corresponding to the largest eigenvalues.

Figure 6.14 illustrates to what extent recovery of the original data is possible when relying on different amounts of eigenvectors. When data is projected to the single eigenvector with the highest eigenvector, all the reconstructed faces look the same except for their level of greyness (see Figure 6.14 (a)). This is not surprising at all, as in this case, we were trying to recover all the faces with the help of one face prototype, i.e. the eigenvector corresponding to the largest eigenvector. As such, the only degree of freedom we have is to control for the grayness of the reconstructed image. As we involve more eigenvectors in Figure 6.14 (b)–(d), the reconstructed images resemble more the original ones from Figure 6.12.

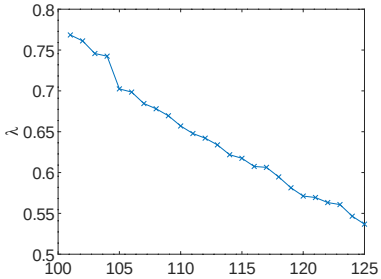
(a) Eigenfaces with eigenvalues ranked between 1 and 25



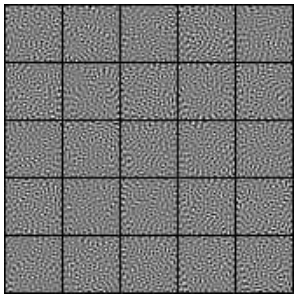
(b) Eigenfaces with eigenvalues ranked between 1 and 25



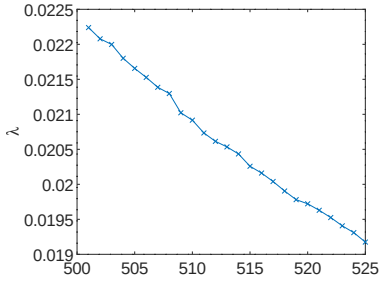
(c) Eigenfaces with eigenvalues ranked between 101 and 125



(d) Eigenfaces with eigenvalues ranked between 101 and 125

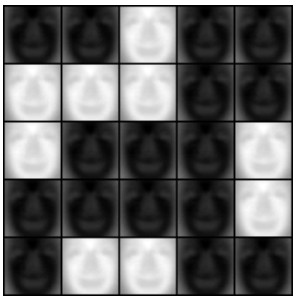


(e) Eigenfaces with eigenvalues ranked between 501 and 525



(f) Eigenfaces with eigenvalues ranked between 501 and 525

Figure 6.13: Differently ranked eigenfaces reflecting the decreasing quality of eigenvectors as their corresponding eigenvalues decrease.



(a) Reconstruction of actual faces based on the single highest ranked eigenvector.

(b) Reconstruction of actual faces based on the 25 highest ranked eigenvectors.

(c) Reconstruction of actual faces based on the 125 highest ranked eigenvectors.

(d) Reconstruction of actual faces based on the 525 highest ranked eigenvectors.

Figure 6.14: Visualizing the amount of distortion when relying on different amount of top-ranked eigenvectors.

6.2.2 *Choosing the reduced dimensionality for PCA*

Our previous example application of utilizing PCA has drawn our attention on how to choose the reduced dimensionality of our data when applying PCA. Suppose, we have access to some originally d–dimensional data, what is the adequate level of reduced dimensionality, $d' < d$ that we shall go for. In other words, how many top eigenvectors shall we rely on during applying PCA on our particular dataset.

Looking at PCA as a form of compressing our data, Figure 6.14 reminds us that the higher level of compression we utilize, i.e. the less eigenvectors we employ, the more reconstruction loss we have to face, meaning that the reconstructed data become less similar to their uncompressed counterparts. Hence, there is a trade-off between how aggressively we decrease the dimensionality of our original data and how much our data with reduced dimensionality would resemble the original data. We can ask ourselves the question, what proportion of the original variance in the dataset gets preserved after we perform PCA on the d' top-ranked eigenvectors.

An important property of every dataset – originating from the diagonalizability of covariance matrices – is that the sum of their

dimension-wise variances is always going to equal the sum of the eigenvalues of their covariance matrix. Based on this useful property of covariance matrices, we can quantify the amount of variance which gets preserved when we perform PCA relying on the d' top-scoring eigenvalues.

Example 6.3. *Based on the previous results, what can we say about the amount of variance preserved when we perform PCA relying on just the single top-scoring eigenvector on the small dataset from Example 6.2, i.e.*

$$M = \begin{bmatrix} 5 & 4 & 0 \\ 4 & 5 & 0 \\ 1 & 1 & 4 \\ 2 & 5 & 5 \end{bmatrix}.$$

In Figure 6.11, we have already seen that matrix M has the eigenvalues $\lambda_1 = 10.65482, \lambda_2 = 2.96484, \lambda_3 = 0.21368$, with a total of 13.83334. What this also tells us, is that the total variance along the three dimensions also equals this quantity. Additionally, if we perform PCA relying on the single highest eigenvector corresponding to the eigenvalue with the largest value, the variance of the transformed data will equal that of $\lambda_1 = 10.65482$, meaning that this way 77.023% of the original variance of the data gets preserved. Shall we perform PCA using the eigenvectors belonging to the top-2 eigenvalues, the preserved variance would be 98.455%, i.e.

$$\frac{\lambda_1 + \lambda_2}{\lambda_1 + \lambda_2 + \lambda_3} = \frac{10.65482 + 2.96484}{10.65482 + 2.96484 + 0.21368} = 0.98455.$$

6.2.3 *A clever trick when $m \ll d$ holds*

It can happen that we have less observations than dimensions, i.e., $m \ll d$. The PCA algorithm for matrix $X \in \mathbb{R}^{m \times d}$ has computational complexity $O(d^3)$ for solving the eigenproblem of the covariance/scatter matrix of X. The calculation of the covariance/scatter matrix requires an additional amount of $O(md^2)$ computation. If d is large, say 10^6 dimensions, this amount of computation seems hopeless to carry out. However, we are not necessarily doomed in such a circumstance either.

Remember that we defined the scatter matrix as

$$S = \sum_{i=1}^{m} \boldsymbol{x}_i \boldsymbol{x}_i^\intercal .$$

Using matrix notation, it can also be equivalently expressed as

$$S = X^\intercal X .$$

Let us now define a somewhat similar matrix

$$T = X X^\intercal ,$$

so that its t_{ij} element store the dot product of observations $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. Notice that the definition of S contains $\boldsymbol{x}_i\boldsymbol{x}_i^\intercal$ (i.e., an outer product resulting in a matrix), whereas the t_{ij} element of T equals $\boldsymbol{x}_i^\intercal\boldsymbol{x}_j$ (i.e., a dot product resulting in a scalar). Nonetheless matrices S and T are defined somewhat similarly, their 'semantics' and dimensions differ substantially as $S \in \mathbb{R}^{d\times d}$ and $T \in \mathbb{R}^{m\times m}$.

Now suppose that $\boldsymbol{u}$ is an eigenvector of T, meaning that there exists some scalar $\lambda \in \mathbb{R}$ such that

$$T\boldsymbol{u} = \lambda\boldsymbol{u}.$$

If we left-multiply both sides of this equality by $X^\intercal$, we get that

$$X^\intercal T\boldsymbol{u} = \lambda X^\intercal\boldsymbol{u},$$

which is simply

$$X^\intercal(XX^\intercal)\boldsymbol{u} = \lambda X^\intercal\boldsymbol{u},$$

according to the definition of T. Due to the associative nature of matrix multiplication, the previous equation can be conveniently rewritten as

$$(X^\intercal X)X^\intercal\boldsymbol{u} = \lambda X^\intercal\boldsymbol{u},$$

which can be equivalently expressed as

$$S(X^\intercal\boldsymbol{u}) = \lambda(X^\intercal\boldsymbol{u})$$

based on how we defined matrix S. This last equation now tells us that it suffices to solve for the eigenvectors of $T \in \mathbb{R}^{m\times m}$, from which we can derive the eigenvectors of $S \in \mathbb{R}^{d\times d}$ by left multiplying them with $X^\intercal$. This way we can reduce the computational need for calculating the eigenvectors of S from $O(d^3)$ to $O(m^3)$, which assuming $m \ll d$ holds, can provide large performance boost in terms of speed.

6.2.4 *Random projections*

It is interesting to note that when we are facing high dimensional data, even though it might sound strange at first sight, random projections of the observations is not a bad idea at all. Of course, the quality of the projections will be somewhat off as if we performed a transformation that is guaranteed to perform the best in some sense. It can be shown by the **Johnson-Lindenstrauss lemma**, that when performing random projections obeying mild assumptions, the distortion of the data will not be that much worse as if we performed a more sophisticated algorithm such as PCA. Here, we are not covering the theory of **random projection**s, however, interested readers can learn about it and its potential applications for image and text data in Bingham and Mannila [2].

[2] Bingham and Mannila 2001

6.3 Singular Value Decomposition

Singular value decomposition (**SVD**) is another approach for performing similar in vein to PCA. A key difference to PCA is that data does not have to be mean centered for SVD, which means that if we work with sparse datasets, i.e. data matrix containing mostly zeros, its sparseness is not 'ruined' by the centering step. We shall note, however, that performing SVD on mean centered data is the same as performing PCA. Let us now focus on SVD in the followings in the general case.

Every matrix $X \in \mathbb{R}^{n \times m}$ can be decomposed into the product of three matrices U, Σ and V such that U and V are orthonormal matrices and Σ is a diagonal matrix, i.e., if some $\sigma_{ij} \neq 0$, it has to follow that $i = j$. The fact that U and V are orthonormal means that the vectors that make these matrices up are pairwise orthogonal to each other and every vector has unit norm. To put it differently, $\boldsymbol{u}_i^\intercal \boldsymbol{u}_j = 1$ only if $i = j$, otherwise $\boldsymbol{u}_i^\intercal \boldsymbol{u}_j = 0$ holds.

Now, we might wonder what these U, Σ, V orthogonal and diagonal matrices are, for which $X = U\Sigma V^\intercal$ holds. In order to see this, let us introduce two matrix products $X^\intercal X$ and $XX^\intercal$. If we express the former based on its decomposed version, we get that

$$X^\intercal X = (U\Sigma V^\intercal)^\intercal (U\Sigma V^\intercal) = (V\Sigma U^\intercal)(U\Sigma V^\intercal) = V\Sigma (U^\intercal U)\Sigma V^\intercal = V\Sigma^2 V^\intercal. \tag{6.18}$$

During the previous derivation, we made use of the following identities:

- the transpose of the product of matrices is the product of the transposed matrices in reversed order, i.e.

$$(M_1 M_2 \dots M_n)^\intercal = M_n^\intercal \dots M_2^\intercal M_1^\intercal,$$

- the transpose of a symmetric and square matrix is itself,
- $M^\intercal M$ equals the identity matrix for any orthogonal matrix M simply by definition.

Additionally, we can obtain via a similar derivation as applied in Eq. (6.18) that

$$XX^\intercal = (U\Sigma V^\intercal)(U\Sigma V^\intercal)^\intercal = (U\Sigma V^\intercal)(V\Sigma U^\intercal) = U\Sigma (V^\intercal V)\Sigma U^\intercal = U\Sigma^2 U^\intercal. \tag{6.19}$$

Now we see two square, symmetric matrices on the left hand sides in Eq. (6.18) and Eq. (6.19). Some of the nice properties that square and symmetric matrices have are that

- their eigenvalues are always real,
- the left and right eigenvalues are going to be the same,
- their eigenvectors are pairwise orthogonal to each other,
- they can be expressed by **eigendecomposition**, i.e., in terms of their eigenvalues and eigenvectors.

MATH REVIEW | EIGENDECOMPOSITION

A matrix M is called **diagonalizable** if there exists some invertible matrix P and diagonal matrix D such that

$$M = PDP^{-1}$$

holds. What this means in other words that M is *similar* to a diagonal matrix. Matrices that are diagonalizable can be also given an eigendecomposition, in which a matrix is expressed in terms of its eigenpairs. The problem of calculating eigenvalues and eigenpairs of matrices have already been covered earlier in Section 3.4.

Figure 6.15: Eigendecomposition

Example 6.4. *Let us find the eigendecomposition of the diagonalizable matrix*

$$M = \begin{bmatrix} 4 & 2 & 1 \\ 5 & 3 & 1 \\ 6 & 7 & -3 \end{bmatrix}.$$

As a first step, we have to find the eigenpairs of matrix M based on the technique discussed earlier in Section 3.4. What we get is that M has the following three eigenvectors

$$\boldsymbol{x_1} = \begin{bmatrix} -0.47 \\ -0.60 \\ -0.64 \end{bmatrix}, \boldsymbol{x_2} = \begin{bmatrix} -0.49 \\ 0.67 \\ 0.55 \end{bmatrix}, \boldsymbol{x_3} = \begin{bmatrix} -0.11 \\ -0.06 \\ 0.99 \end{bmatrix}$$

with the corresponding eigenvalues $\lambda_1 = 7.95, \lambda_2 = 0.15, \lambda_3 = -4.10$.

We know it from earlier that the fact that matrix M has the above eigenpairs means that the following equalities hold:

$$M\boldsymbol{x_1} = \lambda_1 \boldsymbol{x_1},$$
$$M\boldsymbol{x_2} = \lambda_2 \boldsymbol{x_2},$$
$$M\boldsymbol{x_3} = \lambda_3 \boldsymbol{x_3}.$$

A more compact form to state the previous linear systems of equations is the following:

$$M \begin{bmatrix} | & | & | \\ \boldsymbol{x_1} & \boldsymbol{x_2} & \boldsymbol{x_3} \\ | & | & | \end{bmatrix} = \begin{bmatrix} | & | & | \\ \boldsymbol{x_1} & \boldsymbol{x_2} & \boldsymbol{x_3} \\ | & | & | \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_2 \end{bmatrix},$$

which can be conveniently rewritten as

$$M = \begin{bmatrix} | & | & | \\ \boldsymbol{x_1} & \boldsymbol{x_2} & \boldsymbol{x_3} \\ | & | & | \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix} \begin{bmatrix} | & | & | \\ \boldsymbol{x_1} & \boldsymbol{x_2} & \boldsymbol{x_3} \\ | & | & | \end{bmatrix}^{-1}$$

which is exactly analogous to the formula of eigendecomposition. What this means that M can be decomposed into the product of the below matrices

$$\begin{bmatrix} -0.47 & -0.49 & -0.11 \\ -0.60 & 0.67 & -0.06 \\ -0.64 & 0.55 & 0.99 \end{bmatrix} \begin{bmatrix} 7.95 & 0 & 0 \\ 0 & 0.15 & 0 \\ 0 & 0 & -4.10 \end{bmatrix} \begin{bmatrix} -1.07 & -0.66 & -0.16 \\ -0.98 & 0.81 & -0.05 \\ -0.15 & -0.88 & 0.93 \end{bmatrix},$$

with the last matrix being the inverse of the first matrix in the decomposition.

Figure 6.17 illustrates the eigendecomposition of the example matrix M in Octave. As it can be seen, the reconstruction error is practically zero, the infinitesimally small **Frobenius norm** *of the difference matrix between M and its eigendecomposition only arises from numerical errors.*

MATH REVIEW | FROBENIUS NORM

The **Frobenius norm** of some matrix $X \in \boldsymbol{R}^{n \times m}$ is simply the square root of the squared sum of its elements. To put it formally,

$$\|X\|_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{m} x_{ij}^2}.$$

This definition of Frobenius norm makes it a convenient quantity to measure the difference between two matrices. Say, we have some matrix X, the entries of which we would like to approximate as closely as possible by some other matrix (of the same shape) $\tilde{X}$. In such a situation a conventional choice for measuring the goodness of the fit is by calculating $\|X - \tilde{X}\|$. As for a concrete example how to calculate the Frobenius norm of some matrix, take

$$\left\| \begin{bmatrix} 5 & 3 \\ 4 & -6 \end{bmatrix} \right\|_F = \sqrt{5^2 + 3^2 + 4^2 + (-6)^2} = \sqrt{25 + 9 + 16 + 36} = \sqrt{86}.$$

Figure 6.16: Frobenius norm

After this refresher on eigendecomposition, we can turn back to the problem of singular value decomposition, where our goal is to find a decomposition for matrix X in the form of $U\Sigma V^\intercal$, with U and $V^\intercal$ being orthonormal matrices and Σ containing scalars along its main diagonal only.

We have seen it previously that U and V originates from the eigendecomposition of the matrices $XX^\intercal$ and $X^\intercal X$. Additionally, Σ have to consist of the square roots of the corresponding eigenvalues of the eigenvectors in matrices U and $V^\intercal$. Note that the (non-zero) eigenvalues of $XX^\intercal$ and $X^\intercal X$ are going to be the same.

CODE SNIPPET

```
M=[4 2 1; 5 3 1; 6 7 -3];
[eig_vecs, eig_vals] = eig(M);
>>
eig_vecs =

-0.469330  -0.492998  -0.106499
-0.604441   0.671684  -0.064741
-0.643724   0.552987   0.992203

eig_vals =

Diagonal Matrix

7.94734   0         0
0         0.15342   0
0         0        -4.10076

eigen_decomposition = eig_vecs * eig_vals * inv(eig_vecs);
reconstruction_error = norm(M - eigen_decomposition, 'fro')
>>
reconstruction_error =    6.7212e-15
```

Figure 6.17: Performing eigendecomposition of a diagonalizable matrix in Octave

6.3.1 *An example application for SVD – Collaborative filtering*

Collaborative filtering deals with the automatic prediction regarding the interest of a user towards some product/item based on the behavior of the crowd. A typical use case for collaborative filtering is when we try to predict the utility of some product for a particular user such as predicting star ratings a user would give to a particular movie. Such techniques are prevalently applied successfully in **recommendation systems**, where the goal is to suggest items for

users that – based on our predictive model – we hypothesize the user would like.

Recall that user feedback can manifest in multiple forms, star ratings being one of the most obvious and *explicit* form of expressing a user's feeling towards some product. An interesting problem is to build recommender systems exploiting implicit user feedback as well, e.g. from such events that a user stopped watching a movie on some streaming platform. Obviously there could be other reasons for stop watching a movie other than not finding it interesting, and on the other hand just because someone watched a movie from the beginning to its end does not mean that the user found it entertaining. From the above examples, we can see that dealing with implicit user feedback instead of explicit one makes the task of collaborative filtering more challenging.

It turns out that singular value decomposition can be used for solving the above described problem due to its ability of detecting latent factors or concepts in datasets (e.g. a movie rating dataset) and expressing observations with their help. Assume we are given a user–item rating matrix storing ratings of users towards movies they watched where a 5-star rating conveys a highly positive attitude towards a particular movie, whereas a 1 star rating is given by users who really disliked some movie. Table 6.2 includes a tiny example for such a dataset.

	Alien	Rambo	Toy Story
Tom	5	3	0
Eve	4	5	0
Kate	1	0	4
Phil	2	0	5

Table 6.2: Example rating matrix dataset.

If we perform SVD over this rating matrix, we can retrieve an alternative representation of each user and movie according to the latent space. Intuitively, the *latent* dimensions in such an example could be imagined as movie genres, such as *comedy*, *drama*, *thriller*, etc. Taking this view, every entry of the matrix to be decomposed, i.e. the value a particular user gave to a movie in our example, can be expressed in terms of the latent factors. More precisely, a particular rating can be obtained if we take the user's relation towards the individual latent factors and that for the movie as well and weight it by the importance of latent factors (movie genres). In the SVD terminology, we can get these scores from the singular vectors (i.e. the corresponding elements of U and V) and the singular values (i.e. the corresponding value from Σ). The visual representation of the sample dataset from Table 6.2 is included in Figure6.18.

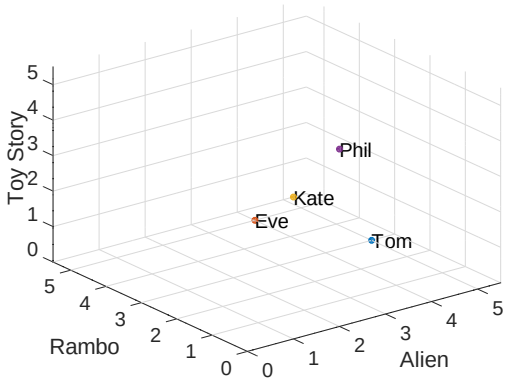


Figure 6.18: Visual representation of the user rating dataset from Table 6.2 in 3D space.

Performing singular value decomposition over the rating matrix from Table 6.2 can be seen below:

$$\begin{bmatrix} 5 & 3 & 0 \\ 4 & 5 & 0 \\ 1 & 0 & 4 \\ 2 & 0 & 5 \end{bmatrix} =$$

$$\begin{bmatrix} -0.63 & 0.22 & 0.73 & -0.25 \\ -0.67 & 0.33 & -0.65 & 0.20 \\ -0.21 & -0.58 & -0.19 & -0.74 \\ -0.33 & -0.72 & 0.08 & 0.59 \end{bmatrix} \begin{bmatrix} 8.87 & 0 & 0 \\ 0 & 6.33 & 0 \\ 0 & 0 & 1.52 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -0.75 & -0.59 & -0.28 \\ 0.06 & 0.36 & -0.93 \\ 0.65 & -0.72 & -0.24 \end{bmatrix}$$

It can be observed in the above example that no matter what the last column in U is, it will not influence the quality of the decomposition since the entire last row of Σ contains zeros. Recall that a matrix has as many singular values as its **rank** r. As a reminder the rank of a matrix is its number of linearly independent column/row vectors. Since $r \leq \min(m, n)$ for any matrix with m rows and n columns, our initial matrix cannot have more than three singular values. In other words, the fourth column in Σ cannot contain any value different from zero, hence the effect of the fourth column in U gets annulled. As such, the decomposition can also be written in a reduced form, making it explicit that our data in this particular case has rank three. That explicit notation is

$$\begin{bmatrix} 5 & 3 & 0 \\ 4 & 5 & 0 \\ 1 & 0 & 4 \\ 2 & 0 & 5 \end{bmatrix} =$$

$$\begin{bmatrix} -0.63 & 0.22 & 0.73 \\ -0.67 & 0.33 & -0.65 \\ -0.21 & -0.58 & -0.19 \\ -0.33 & -0.72 & 0.08 \end{bmatrix} \begin{bmatrix} 8.87 & 0 & 0 \\ 0 & 6.33 & 0 \\ 0 & 0 & 1.52 \end{bmatrix} \begin{bmatrix} -0.75 & -0.59 & -0.28 \\ 0.06 & 0.36 & -0.93 \\ 0.65 & -0.72 & -0.24 \end{bmatrix}$$

We can think of the decomposition in another way as well, i.e.,

$$X = \sum_{i=1}^{rank(X)} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^\intercal.$$

$$8.87 \begin{bmatrix} -0.63 \\ -0.67 \\ -0.21 \\ -0.33 \end{bmatrix} \begin{bmatrix} -0.76 & -0.59 & -0.28 \end{bmatrix} + 6.33 \begin{bmatrix} 0.22 \\ 0.33 \\ -0.58 \\ -0.72 \end{bmatrix} \begin{bmatrix} 0.06 & 0.36 & -0.93 \end{bmatrix} +$$

$$1.52 \begin{bmatrix} 0.73 \\ -0.65 \\ -0.19 \\ 0.08 \end{bmatrix} \begin{bmatrix} 0.65 & -0.72 & -0.24 \end{bmatrix}$$

Basically, this last observation brings us to the idea how we actually use SVD to perform dimensionality reduction. Previously, we said that the last column of U could be discarded as it corresponded to a singular value of zero. Taking this one step further, we can decide to discard additional columns from U and V which belong to some substantially small singular value. Practically, we can think of thresholding the singular values σ, such that whenever it falls behind some value τ, we artificially treat it as if it were zero. We have seen it previously, that whenever a singular value is zero, it cancels the effect of its corresponding singular vectors, hence, we can discard them as well. Based on that observation, we can give a **truncated SVD** of the input matrix X as

$$\tilde{X} = U_k \Sigma_k V_k^\intercal,$$

with $U_k \in \mathbb{R}^{n \times k}, V \in \mathbb{R}^{m \times k}$ being derived from the SVD of X by keeping the k singular vectors belonging to the k largest singular values, and $\Sigma_k \in \mathbb{R}^{k \times k}$ containing these corresponding singular values in the diagonal.

A nice property of the previously described approach is that this way we can control the rank of $\tilde{X}$, i.e. our approximation for X. The rank of $\tilde{X}$ will always be k, denoting the number of singular values that are left as non-zero.

Example 6.5. *Calculate a lower rank approximations of our on-going example movie rating database. Previously, we have seen how it is possible to reconstruct our original data matrix of rank 3, by relying on all three of its singular vectors.*

Now if we would like to approximate the ratings with a rank 2 matrix, all we have to do is to discard the singular vectors belonging to the smallest singular value. This way we get

$$X \approx \tilde{X} = \sum_{i=1}^{2} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^\intercal =$$

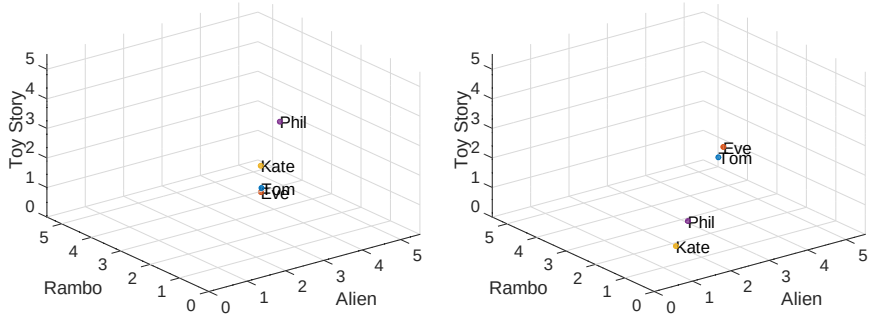
$$= 8.87 \begin{bmatrix} -0.63 \\ -0.67 \\ -0.21 \\ -0.33 \end{bmatrix} \begin{bmatrix} -0.76 & -0.59 & -0.28 \end{bmatrix} + 6.33 \begin{bmatrix} 0.22 \\ 0.33 \\ -0.58 \\ -0.72 \end{bmatrix} \begin{bmatrix} 0.06 & 0.36 & -0.93 \end{bmatrix} =$$

$$= \begin{bmatrix} 4.282 & 3.793 & 0.260 \\ 4.647 & 4.286 & -0.234 \\ 1.190 & -0.210 & 3.931 \\ 1.919 & 0.090 & 5.029 \end{bmatrix}.$$

As we can see, this is a relatively accurate estimate of the originally decomposed rating matrix X, with its elements occasionally overshooting, sometimes underestimating the original values by no more than 0.8 in absolute terms. As our approximation relied on two singular values, $\tilde{X}$ now has a rank of two and is depicted in Figure 6.19(a).

Based on a similar calculation – by just omitting the second term from the previous sum – we get that the rank 1 approximation of X is

$$\sigma_1 \boldsymbol{u}_1 \boldsymbol{v}_1^\intercal = 8.87 \begin{bmatrix} -0.63 \\ -0.67 \\ -0.21 \\ -0.33 \end{bmatrix} \begin{bmatrix} -0.76 & -0.59 & -0.28 \end{bmatrix} = \begin{bmatrix} 4.194 & 3.288 & 1.560 \\ 4.518 & 3.542 & 1.680 \\ 1.419 & 1.113 & 0.528 \\ 2.202 & 1.726 & 0.819 \end{bmatrix}.$$



(a) Reconstuction with 2 singular vectors

(b) Reconstuction with 1 singular vector

Figure 6.19: The reconstructed movie rating dataset based on different amount of singular vectors.

An additional useful connection between singular values of some matrix M and its Frobenius norm is that

$$\|M\|_F^2 = \sum_{i=1}^{rank(M)} \sigma_i^2.$$

The code snippet in Figure 6.20 also illustrates this relation for our running example data matrix M.

This property of the singular values also verifies our choice for discarding those singular values of an input matrix X with the least magnitude. This is because the reconstruction loss – expressed in

CODE SNIPPET

```
M=[5 4 0; 4 5 0; 1 1 4; 2 5 5];
frobenius_norm_sqrd = norm(M, 'fro')^2;
[U,S,V] = svd(M);
singular_vals_sqrd = diag(S).^2;
printf("%f\n", frobenius_norm_sqrd - sum(singular_vals_sqrd))
>> 0.00000
```

Figure 6.20: An illustration that the squared sum of singular values equals the squared Frobenius norm of a matrix.

terms of (squared) Frobenius norm – can be minimized by that strategy. Supposing that the input matrix X has rank r and that the $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_r > 0$ property holds for its singular values, the reconstruction error that we get when relying on a truncated SVD of X based on its top k singular values is going to be

$$\|X - \tilde{X}\|_F^2 = \|U\Sigma V^\intercal - U_k\Sigma_k V_k^\intercal\|_F^2 = \sum_{i=k+1}^{r} \sigma_i^2,$$

meaning that the squared Frobenius norm between the original matrix and its rank k reconstruction is going to be equal to the squared sum of singular values that we made zero. Leaving k singular values non-zero is required to obtain a rank k approximation, and zeroing out the necessary number of singular values with the least magnitude is what makes sense as their squared sum will affect the loss that occurs.

6.3.2 *Transforming to latent representation*

Relying on the SVD decomposition of the input matrix, we can easily place row and column vectors corresponding to real world entities, i.e. users and movies in our running example, into the latent space determined by the singular vectors. Since $X = U\Sigma V^\intercal$ (and $X \approx U_k\Sigma_k V^\intercal = \tilde{X}$ for the lower rank approximation) holds, we also get that $XV = U\Sigma$ (and similarly $XV_k \approx U_k\Sigma_k$).

A nice property is that we are also able to apply the same transformation encoded by V in order to bring a possibly unseen user profile to the latent space as well. Once a data point is transformed in the latent space, we can apply any of our favorite similarity or distance measure (see Chapter 4) to find similar data points to it in the concept space.

Example 6.6. *Suppose we performed SVD already on the small movie rating database that we introduced in Table 6.2. Now imagine that a new user* Anna *shows up who watches the movie* Alien *and gives it a 5-star rating. This means that we have a new user, $\mathbf{x}$ not initially seen in the*

? Data points can naturally be compared based on their explicit representation in the original space. What reasons can you think of which would make working in the latent space more advantageous as opposed to dealing with the original representation of the data points?

rating matrix X. Say, we want to offer Anna users to follow who might have similar taste for movies.

We can use the V_2 for coming up with the rank 2 latent representation of Anna. *Recall that V_2 is the matrix containing the top 2 right singular vectors of X. It means that we can get a latent representation for* Anna *by calculating*

$$\boldsymbol{x}V_2 = \begin{bmatrix} 5 & 0 & 0 \end{bmatrix} \begin{bmatrix} -0.755 & 0.063 \\ -0.592 & 0.361 \\ -0.281 & -0.930 \end{bmatrix} = \begin{bmatrix} -3.777 & 0.313 \end{bmatrix}.$$

We can calculate the latent representation similarly to all the user as

$$XV_2 = \begin{bmatrix} -5.553 & 1.397 \\ -5.982 & 2.058 \\ -1.879 & -3.659 \\ -2.915 & -4.526 \end{bmatrix}.$$

the visualization of which can also be seen in Figure 6.21(a). Now we can calculate the cosine similarity (cf. Section 4.3) between the previously calculated latent vector representation for Anna *and the other users. Cosine similarities calculated between the 2-dimensional latent representation of* Anna *and the rest of the users included in Table 6.3.*

	Tom	Eve	Kate	Phil
Cosine similarity	0.987	0.969	0.382	0.470

Table 6.3: The cosine similarities between the 2-dimensional latent representations of user Anna and the other users.

The cosine similarities in Table 6.3 seem pretty plausible, suggesting that Anna, *who enjoyed watching the movie* Alien, *behaves similar to other users who also enjoyed movies of the same genre as* Alien.

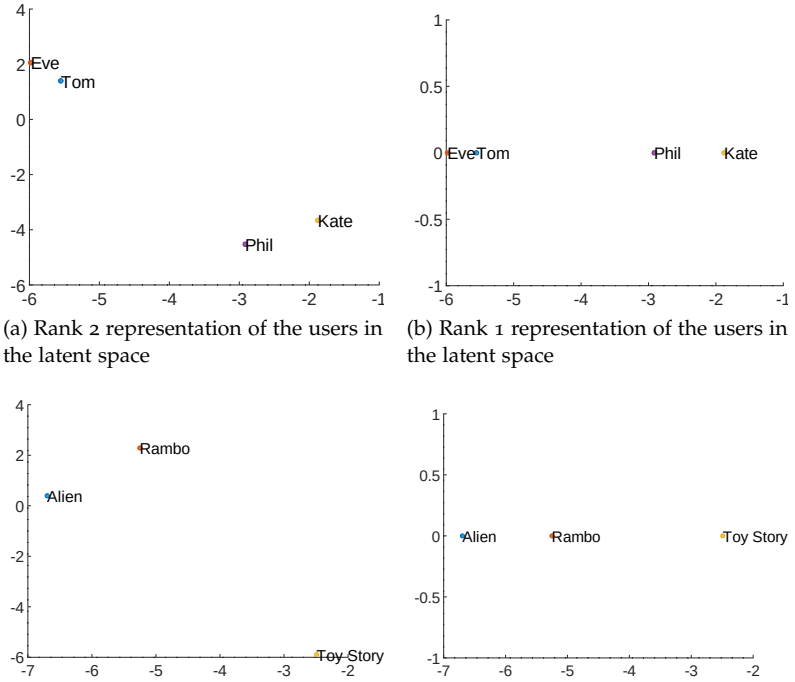
Example 6.7. *Another thing SVD can do for us is to give a predicted ranking of items (movies in our example) a user would give to unrated items based on the latent representation the model identifies. By multiplying the rating profile of a user by V_k followed by a multiplication with $V_k^\intercal$ tells us what would be the most likely ratings of the user with the given rating profile if we simply forget about latent factors (genres) other than the top k most prominent ones.*

For the previous example, this approach would tell us that Anna *is likely to give the ratings included in Table 6.4 when we perform our predictions relying on the top-2 singular vectors of the input rating matrix.*

	Alien	Rambo	Toy Story
Predicted rating	2.872	2.349	0.77

Table 6.4: The predicted rating given by Anna to the individual movies based on the top-2 singular vectors of the rating matrix.

We obtained the predictions in Table 6.4 by multiplying the rating vector of Anna *with $V_2V_2^\intercal$, i.e., $\begin{bmatrix} 5 & 0 & 0 \end{bmatrix} V_2V_2^\intercal$.*



(a) Rank 2 representation of the users in the latent space

(b) Rank 1 representation of the users in the latent space

(c) Rank 2 representation of the movies in the latent space

(d) Rank 1 representation of the movies in the latent space

Figure 6.21: The latent concept space representations of users and movies.

From this, we can conclude that Anna *probably prefers the movie* Alien *the most, something we already could have suspected from the high rating* Anna *gave to that movie. More importantly, the above calculation gave us a way to hypothetize ratings* Anna *would gave to movies not actually rated by her. This way we can argue, that – among the movies she has not rated yet – she would enjoy* Rambo *the most. This answer probably meets our expectations as – based on our commonsense knowledge on movies – we would assume* Rambo *being more similar to the movie* Alien *than* Toy Story.

6.3.3 *CUR decomposition*

The data matrix X that we decompose with SVD is often extremely sparse, meaning that most of its entries are zeroes. Just think of a typical user–item rating matrix in which an element x_{ij} indicates whether user i has rated item j so far. In reality, users do not interact with the majority of the product space, hence it is not uncommon to deal with such matrices the elements of which are dominantly zeroes.

Sparse data matrices are extremely compelling to work with as they can be processed much more effectively relative to dense matrices. This is because we can omit the explicit storage of the zero entries from the matrix. When doing so, the benefit of applying sparse matrix representations is that the memory footprint of the matrix is

not going to be affected directly by its number of rows and columns, instead the memory consumption will be proportional to the number of non-zero elements in the matrix.

A problem with SVD is that the matrices U and V in the decomposition become dense no matter how sparse the matrix that we decomposed is. Another drawback of SVD is that the coordinates in the latent space are difficult to interpret. CUR, offers an effective family of alternatives to SVD. CUR circumvents the above mentioned limitations of SVD by decomposing X into a product of such three matrices C, U and R that vectors comprising matrices C and R originate from the columns and rows of the input matrix X. This behavior ensures that C and R will preserve the sparsity of X. Furthermore, the basis vectors in C and R will be interpretable, as we would know their exact meaning from the input matrix X.

One of the CUR variants works in the following steps:

1. Sample k rows and columns from X with probability proportional to their share from the Frobenius norm of the matrix and let C and R contain these sampled vectors

2. Create $W \in \mathbb{R}^{kxk}$ from the values of X from the intersection of the k selected rows and columns

3. Perform SVD on W such that $W = X\Sigma Y^\intercal$

4. Let $U = Y\Sigma^\dagger X^\intercal$ where $\Sigma^\dagger$ is the (pseudo)inverse of Σ; in order to get the pseudoinverse of a diagonal matrix, all we have to do is to take the reciprocal of its non-zero entries.

Intuitively the share of a vector to the Frobenius norm of the matrix it can be found tells us the relative *importance* of that data point in some sense. The larger fraction of the Frobenius norm can be accounted to a vector, the more larger weight we would like to assign to it. Note that because we sample the vectors with replacement, it can happen that a particular vector (presumably with a larger norm) is sampled multiple times. In order to avoid having multiple copies of the same vector showing up im matrices C and R, a common step is to rescale the sampled vectors. Instead of selecting a row/column vector from the input matrix as is, we instead scale that vector by a factor of $1/\sqrt{kp_i}$, with k referring to the number of row/columns we sample and p_i is the share of the selected vector from the Frobenius norm of the matrix it is sampled from. Note that the scaling factor kp_i is nothing else, but the expected number of times the particular vector gets sampled over k sampling steps. With the above scaling procedure, we can merge together any vector which might be sampled more than once. Based on the peculiarities of the CUR decomposition technique,

it is clear the neither matrix C (corresponding to matrix U from SVD decomposition) nor matrix R (corresponding to matrix V from SVD decomposition) will consist of pairwise orthonormal vectors, instead they will constitute of actual (rescaled) observations, increasing the interpretability of the decomposition.

	Alien	Rambo	Toy Story	P
Tom	5	3	0	$\frac{34}{121}$
Eve	4	5	0	$\frac{41}{121}$
Kate	1	0	4	$\frac{17}{121}$
Phil	2	0	5	$\frac{29}{121}$
P	$\frac{46}{121}$	$\frac{34}{121}$	$\frac{41}{121}$	

Figure 6.22: Example dataset for CUR factorization. The last row/column includes the probabilities for sampling the particular vector.

$$C = \begin{bmatrix} 5 & 0 \\ 4 & 0 \\ 1 & 4 \\ 2 & 5 \end{bmatrix}, U = C^{\dagger}XR^{\dagger} \approx \begin{bmatrix} 0.22 & -0.01 \\ -0.08 & 0.20 \end{bmatrix}, R = \begin{bmatrix} 5 & 3 & 0 \\ 2 & 0 & 5 \end{bmatrix}$$

Multiplying the factors together, we get that

$$CUR = \begin{bmatrix} 5 & 0 \\ 4 & 0 \\ 1 & 4 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} 0.22 & -0.01 \\ -0.08 & 0.20 \end{bmatrix} \begin{bmatrix} 5 & 3 & 0 \\ 2 & 0 & 5 \end{bmatrix} \approx \begin{bmatrix} 5.4 & 4.4 & -0.2 \\ 4.3 & 3.5 & -0.1 \\ 1.1 & -0.4 & 4.0 \\ 2.1 & 0.1 & 4.9 \end{bmatrix}.$$

6.3.4 *Further extensions*

Compact Matrix Decomposition (CMD)[3] and Colibri [4] are extensions of the CUR decomposition. CMD extends CUR in the sense that it avoids the selection of duplicate observations from the decomposition. The Colibri approach further takes care not to include linearly dependent columns in the decomposition resulting in an algorithm which is more efficient compared to its predecessors both in terms of speed and space requirements. A further beneficial property of the Colibri algorithm is that it can be efficiently applied for dynamically changing datasets.

[3] Sun et al. 2008
[4] Tong et al. 2008

Tensors are generalizations of matrices that are allowed to have more than two modes, i.e., instead of having just a 'width' and a 'height', they also come with a 'depth' for instance. Tensors of even higher order can also be thought. As for an illustration of how a tensor might look like, see Figure 6.23.

Performing low-rank decomposition for tensors is an interesting field with lots of potential use cases such as knowledge base comple-

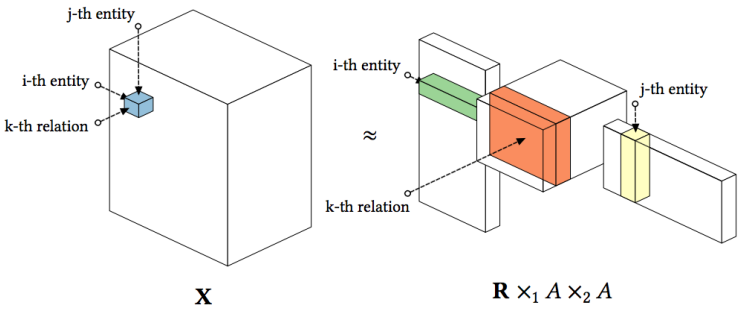


Figure 6.23: An illustration of a tensor.

tion [5]. Despite of its practical utility, an in-depth introduction to the topic of tensors and tensor decomposition is well beyond the scope of this notes. Interested readers are highly encouraged to read the survey of Kolda and Bader [6].

[5] Trouillon et al. 2017, Kazemi and Poole 2018

[6] Kolda and Bader 2009

6.4 *Linear Discriminant Analysis*

Linear discriminant analysis (**LDA**) is a technique which also makes use of the category of the data points they belong for performing dimensionality reduction. More precisely, we are going to assume for LDA that our data points $\boldsymbol{x}_i$ are accompanied by a categorical class label $y_i \in \mathcal{Y}$ that they are characterized by. For the sake of simplicity, we can assume $|\mathcal{Y}| = 2$, that is, every point belongs to either of the positive or negative classes. Having access to the class label of the data points makes two different objectives equally sensible now for reducing the dimensionality of our data points.

On the one hand, it can be argued that points belonging to different classes should be as much separable from each other as possible after dimensionality reduction. What we want in other words, is that points labeled differently mix to the least possible extent. From this perspective, our goal is to find a transformation characterized by $\boldsymbol{w}$ which maximizes the distance between the transformed data points belonging to the different classes. This goal can be equivalently expressed and formalized via relying on the means of the points belonging to the different classes, i.e., $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$. This is due to the fact that applying the same transformation $\boldsymbol{w}$ to all the points will also affect their mean accordingly, i.e., the transformed means are going to be $\boldsymbol{w}^\intercal \boldsymbol{\mu}_1$ and $\boldsymbol{w}^\intercal \boldsymbol{\mu}_2$. The first criteria hence can be expressed as

$$\max_{\boldsymbol{w}} \|\boldsymbol{w}^\intercal \boldsymbol{\mu}_1 - \boldsymbol{w}^\intercal \boldsymbol{\mu}_2\|_2^2 = \max_{\boldsymbol{w}} \boldsymbol{w}^\intercal S_B \boldsymbol{w}, \tag{6.20}$$

where S_B is a rank-1 matrix responsible for characterizing the between-class scatter of the data points according to their original representa-

tion and which can be conveniently calculated in the binary ($|\mathcal{Y}| = 2$) case as

$$S_B = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^\mathsf{T}.$$

In the case, we have more than two classes ($|\mathcal{Y}| > 2$), the between-class scatter matrix is generalized as

$$S_B = \sum_{c=1}^{|\mathcal{Y}|} n_c(\boldsymbol{\mu}_c - \boldsymbol{\mu})(\boldsymbol{\mu}_c - \boldsymbol{\mu})^\mathsf{T},$$

with n_c referring to the number of data points falling into class c, $\boldsymbol{\mu}_c$ being the mean data point calculated from the n_c observations and $\boldsymbol{\mu}$ denoting the mean vector calculated from all the data points irrespective of their class labels.

On the other hand, someone might argue – along the lines of "birds of a feather flock together" – that those points which share the same class label are supposed to be clustered densely after dimensionality reduction is performed. To put it differently, the average distance between the images of the original points within the same category should be minimized. This formally can be quantified with the help of the within-class scatter score between data points. The within-class scatter for data points belonging with class c for a particular projection given by $\boldsymbol{w}$ can be expressed as

$$\tilde{s}_c^2 = \sum_{\{(\boldsymbol{x}_i, y_i)|y_i = c\}} (\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_i - \boldsymbol{w}^\mathsf{T}\boldsymbol{\mu}_c)^2 =$$

$$= \sum_{\{(\boldsymbol{x}_i, y_i)|y_i = c\}} \boldsymbol{w}^\mathsf{T}(\boldsymbol{x}_i - \boldsymbol{\mu}_c)(\boldsymbol{x}_i - \boldsymbol{\mu}_c)^\mathsf{T}\boldsymbol{w} = \boldsymbol{w}^\mathsf{T} S_c \boldsymbol{w},$$

with S_c denoting the scatter matrix calculated over the data points belonging to class c, i.e.

$$S_c = \sum_{\{(\boldsymbol{x}_i, y_i)|y_i = c\}} (\boldsymbol{x}_i - \boldsymbol{\mu}_c)(\boldsymbol{x}_i - \boldsymbol{\mu}_c)^\mathsf{T}.$$

For notational convenience, we shall refer to the sum of within class scatter matrices for class $c = 0$ and $c = 1$ as the aggregated within-class scatter matrix, that is

$$S_W = S_0 + S_1,$$

giving us an overall information on how do the data points differ on average from the mean of the class they belong to.

It turns out that these two requirements often act against each other and the best one can do is to find a trade-off between them instead of performing optimally with respect both of them at the same time. In order to give both of our goals a share in the objective function, the expression that we wish to maximize in the case of

LDA is going to be a fraction. Maximizing a fraction is a good idea in this case, as it can be achieved by a large nominator and a small denominator. Hence the expression we aim at optimizing is

$$\max_{\boldsymbol{w}} \frac{\boldsymbol{w}^\intercal S_B \boldsymbol{w}}{\boldsymbol{w}^\intercal S_W \boldsymbol{w}}, \tag{6.21}$$

with S_B and S_W denoting the between-class and within-class scatter matrices, respectively.

Eq. (6.21) can be maximized if

$$\nabla_{\boldsymbol{w}} \frac{\boldsymbol{w}^\intercal S_B \boldsymbol{w}}{\boldsymbol{w}^\intercal S_W \boldsymbol{w}} = \mathbf{0} \Leftrightarrow (\boldsymbol{w}^\intercal S_B \boldsymbol{w}) \nabla_{\boldsymbol{w}} \boldsymbol{w}^\intercal S_W \boldsymbol{w} = (\boldsymbol{w}^\intercal S_W \boldsymbol{w}) \nabla_{\boldsymbol{w}} \boldsymbol{w}^\intercal S_B \boldsymbol{w} \tag{6.22}$$

is satisfied, which can be simplified as

$$S_B \boldsymbol{w} = \lambda S_W \boldsymbol{w}. \tag{6.23}$$

Upon transitioning from Eq. (6.22) to Eq. (6.23) we made use of the fact that $\nabla_{\boldsymbol{x}} \boldsymbol{x}^\intercal A \boldsymbol{x} = A^\intercal \boldsymbol{x} + A\boldsymbol{x} = (A + A^\intercal)\boldsymbol{x}$ for any vector $\boldsymbol{x}$ and matrix A. In the special case, when matrix A is symmetric – exactly what scatter matrices are – $\nabla_{\boldsymbol{x}} \boldsymbol{x}^\intercal A \boldsymbol{x} = 2A\boldsymbol{x}$ also holds. Eq. (6.23) is pretty much reminds us to the standard eigenvalue problem, except for the fact that there is an extra matrix multiplication on the right hand side of the equation as well. These kind of problems are called **generalized eigenvalue problem**s. There are multiple ways to solve generalized eigenvalue problems. There are more convoluted and effective approaches to solve such problems, but we can also solve them by simply left multiplying both sides with S_W^{-1}, yielding

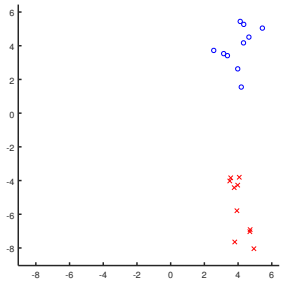
$$S_W^{-1} S_B \boldsymbol{w} = \lambda \boldsymbol{w}$$

that we can regard as a regular eigenproblem. We shall add that in our special case, with only two class labels, we can express also obtain the optimal solution in a simpler form, i.e.

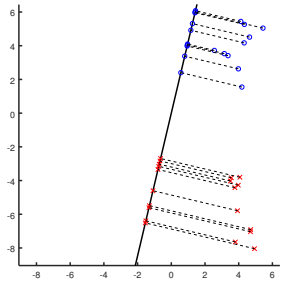
$$\boldsymbol{w}^* = S_W^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2).$$

Example 6.8. *In order to geometrically illustrate the different solutions one would get if the objective was either just the nominator or the denominator of the joint objective function (Eq. (6.21)) let us consider the following synthetic example problem.*

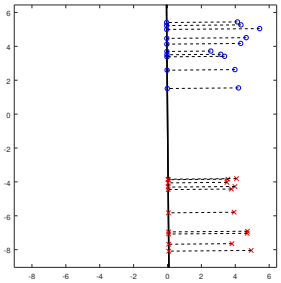
Assume that those data points belonging to the positive class are generated by $\mathcal{N}([4,4],[0.3\ 0;\ 0\ 3])$, that is a bivariate Gaussian distribution with mean vector $[4,4]$ and covariance matrix $[0.3\ 0;\ 0\ 3]$. Likewise, let us assume that the negative class can be described as $\mathcal{N}([4,-5],[0.3\ 0;\ 0\ 3])$, i.e., another bivariate Gaussian which only differs from the previous one in its mean being shifted by 9 units along the second coordinate.



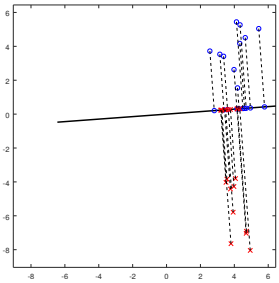
(a) 10 points sampled each from two bivariate normal distributions with means [4,4] and [4,-5].



(b) Optimal projection with respect the joint objective of LDA.



(c) Optimal projection with respect the nominator of the LDA objective.



(d) Optimal projection with respect the denominator of the LDA objective.

Figure 6.24: An illustration of the effect of optimizing the joint fractional objective of LDA (b) and its nominator (c) and denominator (d) separately.

Figure 6.24 (a) includes 10 points sampled from each of the positive and negative classes. Figure 6.24 (b)–(d) contains the optimal projections of this sample dataset when we consider the entire objective of LDA (b), only the term in the nominator (c) and only the term in the denominator (d).

Figure 6.24 (c) nicely illustrates that optimizing for the nominator of the objective of LDA, we are purely focusing on finding a hyperplane which behaves such that the separation between the data points belonging to the different classes get maximized.

Figure 6.24 (d) on the other hand demonstrates that exclusively focusing on the optimization on the denominator of the LDA objective, we obtain a hyperplane which minimizes the scatter for the data points belonging to the same class. At the same time, this approach does not pay any attention for the separation of the data points belonging to the different classes.

The solution seen in Figure 6.24 (b), however, does an equally good job in trying to separate points belonging to distinct categories and minimizing the cumulative within-class scatter.

Table 6.5 contains the distinct parts of the objective function when optimizing for certain parts of the objective in a tabular format. We can see, that – quite unsurprisingly – we indeed get the best objective value for Eq. (6.21) when determining $\boldsymbol{w}^*$ according to the approach of LDA.

Alternative solutions –listed in the penultimate and the last row of Ta-

ble 6.5 – are capable of obtaining better scores for certain parts (either the nominator or the denominator) of the LDA objective, but they fail to do so for the joint, i.e. fractional objective.

? Try calculating the solution with Octave by solving the generalized eigenproblem defined in Eq. (6.23).

CODE SNIPPET

```
# sample 10 examples from the two Gaussian populations
X1 = mvnrnd([4  4], [0.3 0; 0 3], 10);
X2 = mvnrnd([4 -5], [0.3 0; 0 3], 10);
mu1 = mean(X1);
mu2 = mean(X2);
mean_diff = mu1 - mu2;
Sw = (X1 - mu1)' * (X1 - mu1) + (X2 - mu2)' * (X2 - mu2);
w = inv(Sw) * mean_diff;
```

Figure 6.25: Code snippet demonstrating the procedure of LDA.

Objective	$\boldsymbol{w}^*$	$\frac{\boldsymbol{w}^\intercal S_B \boldsymbol{w}}{\boldsymbol{w}^\intercal S_W \boldsymbol{w}}$	$\boldsymbol{w}^\intercal S_B \boldsymbol{w}$	$\boldsymbol{w}^\intercal S_W \boldsymbol{w}$
$\max \frac{\boldsymbol{w}^\intercal S_B \boldsymbol{w}}{\boldsymbol{w}^\intercal S_W \boldsymbol{w}}$	$[-0.23, -0.97]$	**2.32**	85.22	36.80
$\max \boldsymbol{w}^\intercal S_B \boldsymbol{w}$	$[-0.01, \quad 1.00]$	2.29	**90.37**	39.52
$\min \boldsymbol{w}^\intercal S_W \boldsymbol{w}$	$[-1.00, -0.07]$	0.05	0.38	**8.13**

Table 6.5: The values of the different components of the LDA objective (along the columns) assuming that we are optimizing towards certain parts of the objective (indicated at the beginning of the rows). Best values along each column are marked bold.

6.5 Further reading

There is a wide range of further dimensional reduction approaches that are outside the scope of this document. Canonical correlation analysis (CCA) [7] operates over two distinct representations (also often called views) of the dataset and tries to find such transformations, one for each view, which maps the distinct views into a common space such that the correlation between the different views of the same data points gets maximized. Similar to other approaches discussed in this chapter, the problem can be solved as an eigenproblem of a special matrix. Hardoon et al. [2004] provided a detailed overview of the optimization problem and its applications. For a read on the connection of CCA to PCA, refer to the tutorial[8].

[7] Hotelling 1936

[8] Borga 1999

With the advent of deep learning, deep canonical correlation analysis (DCCA) [9] has been also proposed. DCCA learns a transformation expressed by a multi-layer neural network which involves non-linear activation functions. This property of DCCA offers the possibility to learn transformations of more complex nature compared to standard CCA.

[9] Andrew et al. 2013

Contrary to most dimensionality reduction approaches, Locally Linear Embeddings (LLE)[10] provide a non-linear model for dimen-

[10] Roweis and Saul 2000

sionality reduction. LLE focuses on the local geometry of the data points, however, it also preserves the global geometry of the observations in an implicit manner as illustrated by Figure 6.26. LLE makes use of the assumption that even though the global geometry of our data might fail to be linear,individual data points could still be modelled linearly "microscopically", i.e., based on their nearest neighbors.

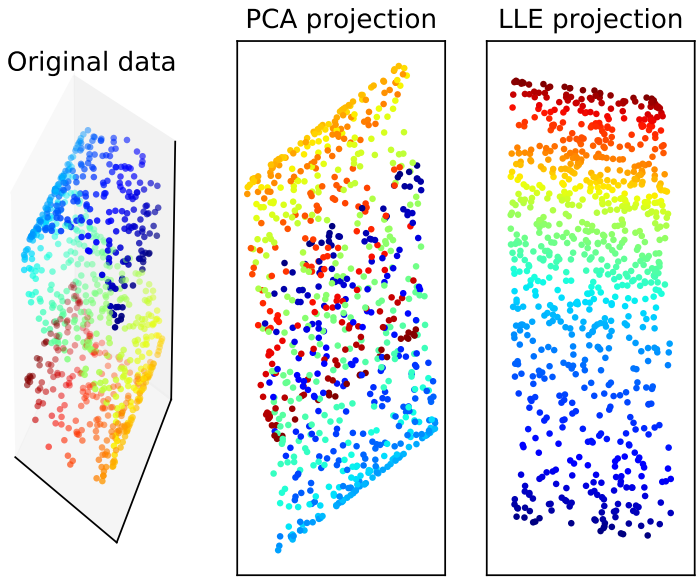


Figure 6.26: Illustration of the effectiveness of locally linear embedding when applied on a non-linear dataset originating from an S-shaped manifold.

Multi dimensional scaling (MDS)[11] tries to give a low-dimensional (often 2-dimensional for visualization purposes) representation of the data, such that the pairwise distances between the data points – assumed to be given as an input – get distorted to the minimal extent. Borg et al. [2012][12] provides a thorough application-oriented overview of MDS.

[11] Kruskal and Wish 1978

[12] Borg et al. 2012

t-Stochastic Neighbor Embedding (t-SNE)[13] aims to find a low-dimensional mapping of typically high-dimensional data points. The algorithm builds upon the pairwise similarity between data points for determining their mapping which can be successfully employed for visualization purposes in 2 or 3 dimensions. t-SNE operates by trying to minimize the Kullback-Leibler divergence between the probability distribution defined over the data points in the original high-dimensional space and their low-dimensional image. t-SNE is known for its sensitivity to the choice of hyperparameters. This aspect of t-SNE is thoroughly analyzed by Wattenberg et al. [2016][14] with ample recommendations and practical considerations for applying t-SNE.

[13] van der Maaten and Hinton 2008

[14] Wattenberg et al. 2016

Most recently, Uniform Manifold Approximation and Projection for Dimension Reduction (UMAP)[15] has been introduced, which

[15] McInnes and Healy 2018

offers increased robustness over t-SNE by assuming that the dataset to be visualized is uniformly distributed on a Riemannian manifold, the Riemannian metric is locally constant and that the manifold is locally connected. For an efficient implementation by the authors of the paper, see `https://github.com/lmcinnes/umap`.

6.6 *Summary of the chapter*

In this chapter we familiarized with some of the most prominent approaches for dimensionality reduction techniques. We derived principal component analysis (PCA) and the closely related algorithm of singular value decomposition (SVD) and their applications. This chapter also introduced CUR decompositions which aims to remedy some of the shortcomings of SVD. The chapter also discussed linear discriminant analysis (LDA) which substantially differs from the other approaches in that it also takes into account the class labels our particular data points belong to. At the end of the chapter, we provided additional references to a series of alternative algorithms that the readers should be able to differentiate and argue for their strength and weaknesses for a particular application.

7 | MINING FREQUENT ITEM SETS

Learning Objectives:

- Learn the concepts related to Frequent item set Mining
- Association rule mining
- Apriori principle
- Park-Chen-Yu algorithm
- FP-Growth and FP trees

Market basket analysis, i.e., analyzing what products are customers frequently purchasing together has enormous business potentials. Supermarkets having access to such information can set up their promotion campaigns with this valuable extra information in mind or they can also decide on their product placement strategy within the shops.

We define a transaction as the act of purchasing multiple items in a supermarket or a web shop. The number of transactions per a day can range between a few hundreds to several millions. You can easily convince yourself about the latter if you think of all the rush going around every year during Black Friday for instance.

The problem of finding item sets which co-occur frequently is called **frequent pattern mining** and our primarily focus in this chapter is to make the reader familiar with the design and implementation of efficient algorithms that can be used to tackle the problem. We should also note that frequent pattern mining need not be interpreted in its very physical sense, i.e., it is possible – and sometimes necessary – to think out-of-the-box and abstractly about the products and baskets we work with. This implies that the problem we discuss in this chapter have even larger practical implications than we might think at first glance.

? Can you think of further non-trivial use cases where frequent pattern mining can be applied?

Example 7.1. *A less trivial problem that can be tackled with the help of frequent pattern mining is that of plagiarism detection. In that case, one would look for document pairs ('item pairs') which use a substantial amount of overlapping text fragments.*

In this setting, whenever a pair of document uses the same phrase in their body, we treat them as a pair of items that co-occur in the same 'market basket'. Market baskets could be hence identified as and labeled by phrases and text fragments included in documents.

Whenever we find a pair of documents being present in the same basket, it means that they are using the same vocabulary. If their co-occurrence exceed some threshold, it is reasonable to assume that this textual overlap is not purely due to chance.

? What kind of information would we find if in the plagiarism detection example we exchanged the roles of documents (items) and text fragments (baskets)?

Exercise 7.1. *Suppose you have a collection of recipes including a list of ingredients required for them. In case you would like to find recipes that are similar to each other, how could you make use of frequent pattern mining?*

There are two main paradigms for performing frequent pattern mining that we review in this chapter. The fist paradigm solves the problem with a *generate and test* philosophy by an iterative approach. That is, starting with an empty set, it constantly tries to expand the already identified frequent item sets and find frequent item sets with an increasing number of items included in them. Such bottom-up approaches inherently require repeated scans over the market basket dataset, which can be time consuming. Other approaches for finding frequent patters in market basket datasets follow a divide-and-conquer philosophy without the need for repeated scans over the dataset. The rest of this chapter elaborates more upon the algorithms and data structures frequently used for frequent pattern mining.

7.1 Important concepts and notation for frequent pattern mining

Before delving into the details of frequent pattern mining algorithms, we define a few concepts and introduce some notations first to make the upcoming discussion easier.

Upon introducing the definitions and concepts, let us consider the example transactional dataset from Table 7.1. The task of frequent pattern mining naturally becomes more interesting and challenging for transactional datasets of much larger size. This is a rather small dataset which includes only five transactions, however, can be conveniently used for illustrative purposes. Note that in the remainder of the chapter, we will use the concepts (market) basket and transaction interchangeably.

Basket ID	Items
1	{milk, bread, salami}
2	{beer, diapers}
3	{beer, wurst}
4	{beer, baby food, diapers}
5	{diapers, coke, bread}

Table 7.1: Example transactional dataset.

7.1.1 Support of item sets

Let us first define the **support** of an item set. Given some item set $\mathcal{I}$, we say that its support is simply the number of transactions $\mathcal{T}_j$ from the entire transactional database $\mathbb{T}$ such that $\mathcal{T}_j \supseteq \mathcal{I}$, that is the market basket with index j contains the item set $\mathcal{I}$. Support is often

reported as a number between 0 and 1, quantifying the proportion of the transactional database which contains item set $\mathcal{I}$.

Note that a basket increases the support of an item set once all elements of the item set are found in a particular basket. Should a single element from an item set be missing from a basket, it no longer qualifies to increase the support for the particular item set. On the other hand, a basket might include arbitrary number of excess items relative to some item set and still contribute to its overall support.

Example 7.2. *Let us calculate the support of the item set {beer} from the example transactional dataset from Table 7.1. The item beer can be found in three transactions (cf. baskets with ID 2,3 and 4), hence its support is also three. For our particular example – when the transactional database contains 5 transactions – this support can also be expressed as* $3/5 = 0.6$.

It is also possible to quantify the support of multi-item item sets. The support of the item set {beer, diapers} is two (cf. baskets with id 2 and 4), or $2/5 = 0.4$ *in the relative sense when normalized by the size of the transactional dataset.*

There is an important property of the support of item sets that we will heavily rely which will ensure the correctness of the Apriori algorithm, being one of the powerful algorithms to tackle the problem of frequent pattern mining. This important property is the **anti-monotonity** of the support of the item sets. What anti-monotonity means in general for some function $f : X \to \mathbb{R}$ is that for any $x_1, x_2 \in X$, that is a pair of inputs from the domain of the function the property

$$x_1 > x_2 \Rightarrow f(x_1) \leq f(x_2)$$

holds, meaning that the value returned by the function for a larger input is allowed to be at most as large as any of the outputs returned by the function for any smaller input.

We define a partial ordering over the subsets of items as depicted in Figure 7.1. According to the partial ordering we say that an item set $\mathcal{I}$ is "larger" than item set $\mathcal{J}$ whenever the relation $\mathcal{I} \supset \mathcal{J}$ holds between the two item sets. Now the anti-monotonity property is naturally satisfied for the support of item sets as the support of a superset of some item set is at most as large as the support of the narrower set.

In the example illustrated by Figure 7.1, item sets $\{b\}, \{c\}, \{b, c\}$ are assumed to be frequent. We indicate the fact that these item sets are considered frequent by marking them with red. Note how the anti-monotone property of support is reflected graphically in Figure 7.1 as all the proper subsets of the frequent item sets are always frequent as well. If this were not the case, that was a violation of the anti-monotone property.

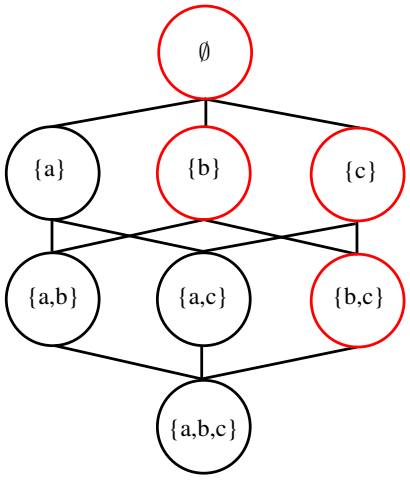


Figure 7.1: An example Hasse diagram for items a, b and c. Item sets marked by red are frequent.

7.1.2 *Association rules*

The next important concept is that of association rules. From a market basket analysis point of view an **association rule** is a pair of (disjoint) item sets, $(\mathcal{X}, \mathcal{Y})$ such that the purchase of item set $\mathcal{X}$ makes the purchase of item set $\mathcal{Y}$ likely. It is notated as $\mathcal{X} \Rightarrow \mathcal{Y}$.

In order to quantify the strength of an association rule, one can calculate its **confidence**, i.e.,

$$c(\mathcal{X} \Rightarrow \mathcal{Y}) = \frac{support(\mathcal{X} \cup \mathcal{Y})}{support(\mathcal{X})},$$

that is the number of transactions containing all of the items present in the association rule, divided by the number of transactions that include at least the ones on the left hand side of the rule (and potentially, but not mandatorily any other items, including the ones on the right hand side of the association rule). What confidence intuitively quantifies for an association rule is a conditional probability, i.e., it tells us the probability that a basket would contain item set $\mathcal{Y}$ given that the basket already contains item set $\mathcal{X}$.

Example 7.3. *Revisiting the example transactional dataset from Table 7.1, let us calculate the confidence of the association rule $\{beer\} \Rightarrow \{diaper\}$. In order to do so we need the support of the item pair {beer, diapers} and that of the single item on the left hand side of the association rule, i.e., {*beer*}.*

Recall that these support values are exactly the ones we calculated in Example 7.2 that is

$$c(\{beer\} \Rightarrow \{diapers\}) = \frac{support(\{beer, diapers\})}{support(\{beer\})} = 2/3.$$

Recall that unlike conditional probabilities are not symmetric, i.e., $P(A|B) = P(B|A)$ need not be the case by definition, the same applies

for the confidence of item sets, meaning that

$$c(\mathcal{X} \Rightarrow \mathcal{Y}) = c(\mathcal{Y} \Rightarrow \mathcal{X})$$

***does not** necessarily hold.*

As an example to see when the symmetry breaks, calculate the confidences of the association rules

$$\{bread\} \Rightarrow \{milk\} \text{ and } \{milk\} \Rightarrow \{bread\}.$$

Also note that association rules can have multiple items on either on their sides, meaning that association rules of the form

$$\{beer\} \Rightarrow \{diapers, babyfood\}$$

are totally legit ones.

7.1.3 The interestingness of an association rule

One could potentially think that association rules with high confidence are needlessly useful. This is not necessarily the case, however. Just imagine the simple case when there is some product A which simply gets purchased by every customer. Since this product can be found in every market basket, no matter what product B we choose for, the confidence of the association rule $c(B \Rightarrow A)$ would also be inevitably 1.0 for any product B.

In order to better access the usefulness of an association rule, we need to devise some notion of true interestingness for the association rules. There exists a variety of such interestingness measures. Going through all of them and detailing their properties is beyond our scope, here we just simply mention a few of the possible ways to quantify the interestingness of an association rule.

A simple way to measure how interesting an association rule $A \Rightarrow B$ is to calculate the so-called **lift** of the association rule by the formula

$$\frac{c(\mathcal{A} \Rightarrow \mathcal{B})}{s(\mathcal{B})},$$

with $c(\mathcal{A} \Rightarrow \mathcal{B})$ and $s(\mathcal{B})$ denoting the confidence of the association rule and the relative support of item set $\mathcal{B}$, respectively. Taking into consideration that the confidence of an association rule can be regarded as a conditional probability of purchasing item set $\mathcal{B}$ given that item set $\mathcal{A}$ had been purchased, and that the relative support of an item set is nothing but the probability of purchasing that given item set $\mathcal{A}$, it is easy to see that the lift of an association rule can be rewritten as

$$\frac{P(A, B)}{P(A)P(B)},$$

with $P(A, B)$ indicating the joint probability of buying both item sets $\mathcal{A}$ and $\mathcal{B}$ simultaneously, $P(A)$ and $P(B)$ referring to the marginal probability of purchasing item sets A and B, respectively. What it means in the end that the lift of a rule investigates to what extent is the purchase of item set $\mathcal{A}$ is independent from that of item set $\mathcal{A}$. A lift value of 1 means that item sets $\mathcal{A}$ and $\mathcal{B}$ are purchased independent from each other. Larger lift values mean a stronger connection between item sets $\mathcal{A}$ and $\mathcal{B}$.

We get a further notion of interestingness for an association rule if we calculate

$$i(\mathcal{A} \Rightarrow \mathcal{B}) = c(\mathcal{A} \Rightarrow \mathcal{B}) - s(\mathcal{B}),$$

where $c(\mathcal{A} \Rightarrow \mathcal{B})$ denotes the confidence of the association rule and $s(\mathcal{B})$ marks the relative support for the item set on the right hand side of the association rule. Unlike lift, this quantity can take negative value, once the condition $s(\mathcal{B}) > c(\mathcal{A} \Rightarrow \mathcal{B})$ holds. This happens when item set $\mathcal{B}$ is less frequently present among such baskets that contain item set $\mathcal{A}$ compared to the overall frequency of the presence of item set $\mathcal{B}$ (irrespective of item set $\mathcal{A}$). A value of zero for that value means that we see item set $\mathcal{B}$ just as frequently in those baskets that contain item set $\mathcal{A}$ as well than in any basket not necessarily containing item set $\mathcal{A}$ in general. A positive value on the other hand means, that the presence of item set in a basket makes the presence of item set $\mathcal{B}$ more likely compared to the case when we do not know if $\mathcal{A}$ is also present in the basket.

7.1.4 *The cardinality of potential association rules*

In order to illustrate the difficulty of the problem we try to solve from a combinatorial point of view, let us quantify the number of possible association rules that one can construct out of d products. Intuitively, in an association rule every item can be either absent or present in the left hand side or the right hand side of the rule. This means that for every item, there are three possibilities in which it can be involved in an association rule, meaning that there are exponentially many, i.e., $O(3^d)$ potential association rules that can be assembled from d distinct items.

Note, however, that the quantity 3^d is an overestimation towards the true number of *valid* association rules, in which we expect both sides to be disjoint and non-empty. By discounting for the invalid association rules and utilizing the equation

$$(1+x)^d = \sum_{j=1}^{d} \binom{d}{j} x^{d-j} + x^d,$$

we get for the exact number of valid association rules to be

$$\sum_{i=1}^{d}\left[\binom{d}{i}\sum_{j=1}^{d-i}\binom{d-i}{j}\right]=\sum_{i=1}^{d}\binom{d}{i}(2^{d-i}-1)=$$

$$=\sum_{i=1}^{d}\binom{d}{i}2^{d-i}-\sum_{i=1}^{d}\binom{d}{i}=(3^d-2^d)-(2^d-1)=3^d-2^{d+1}+1,$$

with $3^d - 2^{d+1} + 1 = O(3^d)$. This means that although our original answer was not correct in the strict sense – because it also included the number of invalid association rules – it was correct in the asymptotic sense.

Based on the above exact results, we can formulate as many as 18,660 possible association rules even when there are just $d = 9$ single items to form association rules from. This quick exponential growth in the number of potential association rules is illustrated in Figure 7.2, making it apparent that without efficient algorithms finding association rules would not practically be feasible. Since association rules can be created by partitioning frequent item sets, it is of upmost importance that we could find frequent item sets efficiently in the first place.

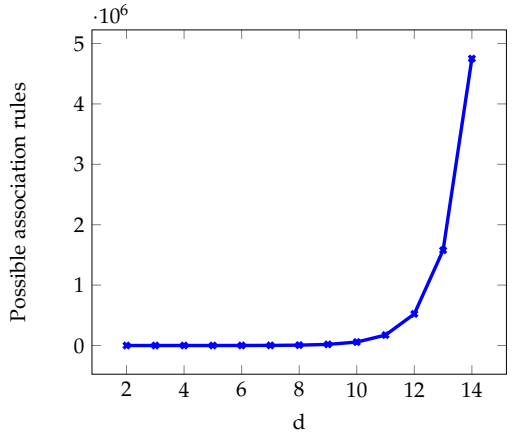


Figure 7.2: Illustration of the distinct potential association rules as a function of the different items/features in our dataset (d).

7.1.5 *Special subtypes of frequent item sets*

Before delving into the details of actual algorithms which efficiently determine frequent item sets, we first define a few important special subtypes of item sets. These special classes of item sets which are beneficial because they allow for a compressed storage of frequent item sets that can be found in some transactional dataset.

- A **maximal frequent item set** is such that all of its supersets are not frequent. To put it formally, an item set I is maximal frequent

if the property

$$\{I | I \text{ frequent } \wedge \nexists \text{ frequent } J \supset I\}$$

holds for it.

- A **closed item set** is such an item set that none of its supersets have a support equal to it, or to put it differently, all of its supersets have a strictly smaller support compared to it. Formally stating this property

$$\{I | \nexists J \supset I : s(J) = s(I)\}$$

has to hold for an item set I to be closed.

- A **closed frequent item set** is simply an item set which is closed in the above sense and which has a support exceeding some previously defined frequency threshold τ.

It can be easily seen that maximal item sets always need to be closed as well. This statement can be verified by contradiction. That is if we suppose that there exists some item set I which is maximal, but which is not closed, we get to a contradiction. Indeed, if item set I is not a closed item set, then it means that there is at least one such superset $J \supset I$ with the exact same support, i.e., $s(I) = s(J)$. Now, since I is a frequent item set based on our initial assumption, so does J because we have just seen that it would have the same support as I.

This, however, contradicts to the assumption of I being a maximal frequent item set, since there exists no superset for maximal frequent item sets that would be frequent as well. Hence maximality of an item set implies its closed property as well. Figure 7.3 summarizes the relation of the different subtypes of frequent item sets.

As depicted by Figure 7.3, maximal frequent item sets holds for just a privileged set of frequent item sets, i.e., those special ones that are located at the **frequent item set border**, also referred to as the **positive border**. Item sets that are located in the positive border are extremely important, since storing only these item sets is sufficient to implicitly store all the frequent item sets. This is again assured by the anti-monotone property of the support of item sets, i.e., all proper subsets of the item sets on the positive border needs to be frequent as well because they have at least the same or even higher support as those item sets present in the positive border.

Additionally, by the definition of maximal frequent item sets, it is also the case that none of their supersets is frequent, hence maximal frequent item sets are indeed sufficient to be stored for implicitly storing all the frequent item sets. Storing maximal frequent item sets, however, would not allow us to reconstruct the supports of all the

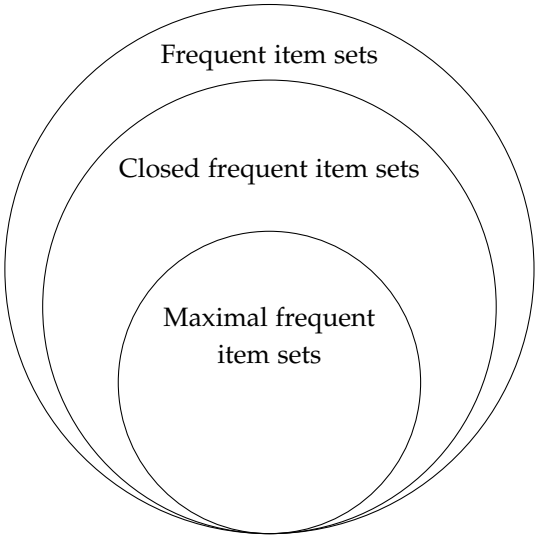


Figure 7.3: The relation of frequent item sets to closed and maximal frequent item sets.

frequent item sets. If it is also important for us that we could tell the exact support for all the frequent item sets, then we also need to store all the closed frequent item sets.

Note, however, that the collection of closed frequent item sets is still narrower than those of frequent item sets (cf. Figure 7.3). Hence, storing closed frequent item sets alone also implicitly stores all the frequent item sets in a compressed form together with their support.

Example 7.4. *Table 7.2 contains a small transactional dataset alongside with the categorization of the different item sets. In this example, we take the minimum support for regarding an item set to be frequent as* 3*. We can see as mentioned earlier that whenever a dataset qualifies being maximal frequent, it always also holds that the given dataset is closed simultaneously.*

Additionally, we can see as well that taking all the proper subsets of the maximal frequent item sets identified in the transactional dataset, we can generate all further item sets that are frequent as well, i.e., the proper subsets of item sets $\{A, B\}$ *and* $\{B, C\}$*, it follows that the individual singleton item sets –* $\{A\}$*,* $\{B\}$ *and* $\{C\}$ *– are also above our frequency threshold that was set to* 3*.*

7.2 *Apriori algorithm*

The Apriori algorithm is a prototypical approach for frequent item set mining algorithms that are based on **candidate set generation**. The basic principle of such algorithms is that they iteratively generate item sets which potentially meet the predefined frequency threshold, then filters them for those that are found to be truly frequent. This iterative procedure is repeated until it is possible to set up a non-

Table 7.2: Compressing frequent item sets — Example (t = 3)

Transaction ID	Basket
t1	{A,B,C}
t2	{A,B,C}
t3	{B,C}
t4	{A,B}
t5	{A,B}

(a) Sample transactional dataset

Item	Frequency	Maximal	Closed	Closed frequent
A	4	No	No	No
B	5	No	Yes	Yes
C	3	No	No	No
AB	4	Yes	Yes	Yes
AC	2	No	No	No
BC	3	Yes	Yes	Yes
ABC	2	No	Yes	No

(b) The characterization of the possible item sets

empty set of candidate item sets.

During the generation phase, we strive for the lowest possible false positive rate, meaning that we would like to count the actual support for the least amount of such item sets which are non-frequent in reality. At the same time, we want to avoid false negatives entirely, meaning that we would not like to erroneously miss investigating whether an item set is frequent if it has such a support which makes it a frequent item set.

At first glance, it sounds like a chicken and egg problem, since the only way to figure out if an item set is frequent or not is to count its support. Hence the idea behind candidate set generating techniques is to rule out as many provably infrequent item sets from the scope of candidates without ruling out any candidate that we were not supposed to do so. We can achieve this by checking certain necessity conditions for item sets before treating them as frequent item set candidates.

A very natural necessity condition follows from the anti-monotonic property of the support of item sets. This necessity condition simply states that in order an item set to be potentially frequent, all its proper subsets also need to be frequent as well.

This suggests us the strategy for regarding all the single items as potentially frequent candidate sets upon initialization and gradually filtering and expanding this initial solution in an iterative manner. Let us denote our initial candidates of frequent item sets compris-

ing of single element item sets as C_1. More generally, let C_i denote candidate item sets of cardinality i.

Once we count the true support for all of the single item candidates in C_1 by iterating through the transactional dataset, we can retain the set of those item singletons that indeed meet our expectations with respect their support being greater than or equal to some predefined frequency threshold t. This way, we can get to the set of truly frequent one-element item sets such that $F_1 = \{i | i \in C_1 \wedge s(i)\} \subset C_1$. The set F_1 is helpful for obtaining frequent item sets of larger cardinality, i.e., according to the anti-monotone property of support, only such item sets have a non-zero chance of being frequent that are supersets of a frequent set in F_1.

In general, once the filtered set of truly frequent item sets of cardinality $k-1$ is obtained, it helps us to construct the set of frequent candidate item sets of increased cardinality k. We repeat this in an ongoing fashion as indicated in the pseudocode provided in Algorithm 2.

Algorithm 2: Pseudocode for calculation of frequent item sets

Require: set of possible items U, transactional database $\mathbb{T}$, frequency threshold t
Ensure: frequent item sets
1: $C_1 := \mathcal{U}$
2: Calculate the support of C_1
3: $F_1 := \{x | x \in C_1 \wedge s(x) \geq t\}$
4: **for** $(k = 2; k < |\mathcal{U}| \&\& F_{k-1} \neq \emptyset; k++)$ **do**
5: Determine C_k based on F_{k-1}
6: Calculate the support of C_k
7: $F_k := \{X | X \in C_k \wedge s(X) \geq t\}$
8: **end for**
9: **return** $\cup_{i=1}^{k} F_i$

7.2.1 *Generating the set of frequent item candidates C_k*

There is one additional important issue that needs to be discussed regarding the details of the Apriori algorithm. That is, how to efficiently generate the candidate item sets C_k in line 5 of Algorithm 2. As it has been repeatedly stated, when setting up frequent item set candidates of cardinality k, the previously calculated supports of the item sets with smaller cardinalities impose necessity conditions that we can rely on for reducing the number of candidates to be generated.

Our goal is then to be as strict in the composition of C_k as possible, whereas not being stricter than what is necessary. That is, we should

not fail to include such an item set in C_k that is frequent in reality, however, it would be nice to see in the end as few excess candidate item sets as possible. In other terminology, also used previously in Chapter 4, we would like to keep false positive errors low, while not tolerating false negative error at all.

We know that in order some item set of cardinality k to have a non-zero probability for being frequent, it has to hold that all item sets of cardinality $k-1$ that we can form by leaving out just a single item also has to be frequent. This requirement follows from the anti-monotonicity of the support. Based on this necessity condition, we could always check all the k proper subsets that we can form from some potential candidate item set of cardinality k by leaving out just one item at a time.

This strategy is definitely the most informed one in the sense that it relies on all the available information that can help us to foresee if a candidate item set cannot be frequent. This well-informedness, however, comes at a price. Notice how this approach scales linearly in the size of the candidate item set we are about to perform a validation for. That is, the larger item sets we would like to perform this preliminary sanity check, i.e., to see if it makes sense at all to treat it as a candidate item set, the more examinations we have to carry out, resulting in more computation.

For the above reasons, there is a computationally cheaper strategy which is typically preferred in practice for generating C_k, i.e., frequent item set candidates of cardinality k. This computationally more efficient strategy, nonetheless has a higher tendency of generating false positive candidates, i.e., candidate item sets which would eventually turn out to be infrequent after we calculate their support. For this computationally simpler strategy of constructing candidate item sets, there need to be an ordering defined over the individual items.

? In what way do you think it makes the most sense to define the order of the items and why it is so?

Once we have an ordering defined over the items found in the dataset, we can set up a simplified criterion towards creating a candidate item set $\mathcal{I} \in C_k$. Note that since there is an ordering for the items to be found in item sets, we can now think of item sets as ordered character sequences in which there is a fixed order in which two items can follow each other.

According to the simplified criterion, it suffices to consider those item sets of cardinality k as being potentially frequent, the ordered string representation is such that when omitting either its last or penultimate character leaves us with two string representations of such item sets of cardinality $k-1$ which have been already verified to be frequent as well.

Note that this simplified strategy is less stringent as opposed to exhaustively checking whether all the $k-1$-element proper subsets

are frequent for an item set consisting of k items. The advantage of the simpler strategy for generating C_k is that the amount of computation it requires does not depend on the cardinality of the item sets we are generating as it always makes a decision on just exactly two proper subsets of a potential candidate item set. This strategy is hence has $O(1)$ computational requirement as opposed to the more rigorous – albeit more expensive ($O(k)$) – exhaustive approach which requires all the proper subsets of a candidate item set to be frequent as well.

Note that the $O(k)$ exhaustive strategy for generating elements of C_k subsumes the check-ups carried out in the computationally less demanding $O(1)$ approach, i.e., the exhaustive strategy performs all the sanity checks involved in our simpler generation strategy. We should add that the $O(1)$ approach we proposed involves a somewhat arbitrary choice for checking a necessity criteria towards an item set to be potentially frequent.

We said that in the simplified approach we regard an item set $\mathcal{I}$ a potentially frequent candidate if its proper subsets that we get by leaving out either the last or the penultimate item from it are also frequent. Actually, – unless nothing is known about the ordering over the items within item sets – this criterion could be easily replaced with the one which treats some item set $\mathcal{I}$ to be a potentially useful candidate, if the item sets that we get by omitting its **first** and last elements are known to be frequent.

In summary, it plays no crucial role in the simplified strategy for generating C_k which two proper subsets do we require for a potential candidate $\mathcal{I}$ to be frequent, what was important is that instead of checking for all the proper subsets of $\mathcal{I}$, we opted for verifying the frequent nature of a fixed sized sample of the $k-1$-item subsets of $\mathcal{I}$. In the followings, nonetheless we will stick to the common implementation of this candidate generating step, i.e., we would check the frequency of those proper subsets of some potential frequent candidate item set $\mathcal{I} \in C_k$ which match on their $k-2$-length prefix in their ordered representations and only differ on their very last items.

Example 7.5. *Suppose that our transactional database contains items labeled by the letters of alphabet from 'a' to 'h'. Let us further assume that the ordering that is required by the simplified candidate nominating strategy follows the natural alphabetical order of letters. Note, that in general it might be a good idea to employ 'less natural' ordering of items, however, we will assume here the usage of alphabetical ordering for simplicity.*

Suppose that the Apriori algorithm have identified

$$F_3 = \{abd, abe, beh, dfg, acd\}$$

as the set of frequent item sets of cardinality 3. How would we determine C_4

then?

One rather expensive way of doing so would try all the possible $\binom{|F_3|}{2}$ *different ways to combine the item sets in* F_3 *and see which of those form an item set consisting of four items when merged together. If we followed this path, we would obtain* $C_4 = \{abde, abcd, abeh\}$*. This strategy would, however, require too much work and also leave us with an excessive amount of false positive item sets in* C_4 *which will not be frequent item set after checking their support.*

Yet another – however rather expensive – approach would check for all the proper subsets of potential item quadruplets if they can be found among F_3*. In that case, we would end up treating* $C_4 = \emptyset$*. This result is very promising as it gives us the lowest possible candidates to check for, and eventually the algorithm could terminate. The downside of this strategy, however, is that it requires checking all the four proper subsets of an item set* $\mathcal{I}$ *before it could get into* C_4*.*

Our simpler approach, that checks if an item set to be included among the candidates in constant time provides a nice trade-off between the two strategies illustrated earlier. We would then combine only such pairs of item sets from F_3 *which match on their first two items and only differ on their last item. This means that if we follow this strategy, we would have* $C_4 = \{abde\}$*.*

Exercise 7.2. *Determine* C_4 *based on the same* F_3 *as in Example 7.5 with the only difference that this time the ordering over our item set follows* $b > c > d > a > g > f > e > h$*.*

7.2.2 Storing the item set counts

Remember that upon the determination of F_k, i.e., the truly frequent item sets of cardinality k, we need to iterate through our transactional dataset $\mathbb{T}$ and allocate a separate counter for each item set in F_k. This is required so that we can determine

$$F_k = \{\mathcal{I} | \mathcal{I} \in C_k \wedge s(\mathcal{I}) \geq t\} \subseteq C_k,$$

the set of those item sets which is a subset of our set of candidate item sets of cardinality k, such that their support is at least some predefined frequency threshold t.

As such, another important implementational detail regarding the Apriori algorithm is how to efficiently keep track of the supports of our candidate item sets in C_k. These counts are important, because we need them in order to formulate $F_k \subseteq C_k$. Recall that the way determine C_k is via the combination of certain pairs of F_{k-1}. This means that during iteration k, we have $\binom{|F_{k-1}|}{2}$ potential counters to keep track for obtaining $F_k \subseteq C_k$, which can be quite a resource-demanding task to do. The amount of memory we need to allocate

for the counters is hence $|C_k| = O(|F_{k-1}|^2)$ as there are this many candidate item sets which we believe to have a non-zero chance of being frequent.

As mentioned earlier, our candidate generation strategy necessarily contains false positives, i.e., such candidate item sets that turn out to be infrequent. Not only our candidates contain item sets that would be proven as being infrequent in reality, it will most likely contain a large proportion of candidate item sets which has an actual support of zero. This means that we potentially allocate some – typically quite a large – proportion of the counters just to remain zero throughout the time, which sounds like an incredible waste of our resources!

To overcome the previous phenomenon, counters are typically create in an on-line fashion. This means that we do not allocate the necessary memory for all the potentially imaginable item pairs in advance, but create a new counter for a pair of item set the first time we see them co-occurring in the same basket.

This solution, however, requires additional memory usage, i.e., since we are not explicitly storing a counter for every combination of item sets from F_{k-1}, we now also need to additionally store alongside our counters the extra information which helps us to identify which element of C_k the particular counter is reserved for. We identify an element from C_k as a pair of indices (i, j) such that

$$C_k = F_{k-1}^{(i)} \cup F_{k-1}^{(j)},$$

that is (i, j) identifies those item sets from F_{k-1} that when merged together exactly results in C_k.

When we store counters in this on-line fashion, we thus need three times the amount of item pairs from F_{k-1} that co-occur in the transactional dataset at least once. Recall that although the explicit storage would not require these additional pairs of indices per non-zero counters, it would nonetheless require $\binom{|F_{k-1}|}{2}$ counters in total. Hence, whenever the number of item sets with cardinality k and a non-zero presence in the transactional dataset is below $\frac{1}{3}\binom{|F_{k-1}|}{2}$, we definitely win by storing the counters for our candidates in the on-line manner as opposed to allocating a separate counter to all the *potential* item sets of cardinality k.

There seems to be a circular referencing in the above criterion, since the only way we could figure out whether the given relation holds is if we count all the potential candidates first. As a rule of thumb, we can typically safely assume that the given relation holds. Moreover, there could be special circumstances when we can be absolutely sure in advance that the on-line bookkeeping is guaranteed to pay off. Example 7.6 contains such a scenario.

Example 7.6. *Suppose we have a transactional database with* 10^7 *transactions and* $200,000$ *frequent items, i.e.,* $|\mathbb{T}| = 10^7$ *and* $|F_1| = 2 * 10^5$*. We would hence need* $\binom{|F_1|}{2} \approx 2 * 10^{10}$ *counters if we wanted to keep explicit track of all the possible item pairs.*

Assume that we additionally know it about the transactional dataset that none of the transactions involve more than 20 products, i.e., $|t_i| \leq 20$ *for all* $t_i \in \mathbb{T}$*. We can now devise an upper bound on the maximal amount of co-occurring item pairs. If this upper bound is still less than one third of the counters needed for the explicit solution, then it is guaranteed that the on-line bookkeeping is the proper way to go.*

If we imagine that every basket includes exactly 20 items (instead of at least 20 items), we get an upper bound on the number of item pairs included in a single basket to be $\binom{20}{2} = 190$*. Making the – rather unreasonable – assumption that item pairs are never repeated in multiple baskets, there are still no more than* $190 * 10^7$ *item pairs that could potentially have a non-zero frequency. This means that we would require definitely no more than* $6 * 10^9$ *counters and indexes in total when choosing to count co-occurring item sets with the on-line strategy.*

Transaction ID	Basket
t1	{1,3,4}
t2	{1,4,5}
t3	{2,4}
t4	{1,4,6}
t5	{1,6}
t6	{2,3}
t7	{1,4,6}
t8	{2,3}

(a) Sample transactional dataset

1	2	3	4	5	6
1	0	1	1	0	0
1	0	0	1	1	0
0	1	0	1	0	0
1	0	0	1	0	1
1	0	0	0	0	1
0	1	1	0	0	0
1	0	0	1	0	1
0	1	1	0	0	0

(b) The corresponding incidence matrix.

Table 7.3: Sample transactional dataset (a) and its explicit and its corresponding incidence matrix representation (b). The incidence matrix contains 1 for such combinations of baskets and items for which the item is included in the particular basket.

7.2.3 *An example for the Apriori algorithm*

We next detail the individual steps involved during the execution of the Apriori algorithm for the sample dataset introduced in Table 7.3. We use three as the threshold for the minimum support frequent item sets are expected to have.

? Can you suggest a vectorized implementation for calculating the support of item sets up to cardinality 2?

1. The first phase of the Apriori algorithm treats all the items as potentially frequent ones, i.e., we have $C_1 = \{1, 2, 3, 4, 5, 6\}$. This means that we need to allocate a counter for each of the items in C_1. The algorithm next iterates through all the transactions and keeps track of the number times an item is present. Figure 7.4 includes an example code snippet for calculating the support

CODE SNIPPET

```
baskets=[1 1 1, 2 2 2, 3 3, 4 4 4, 5 5, 6 6, 7 7 7, 8 8];
items = [1 3 4, 1 4 5, 2 4, 1 4 6, 1 6, 2 3, 1 4 6, 2 3];
dataset = sparse(baskets, items, ones(size(items)));
unique_counts=zeros(1, columns(dataset));
for b=1:rows(dataset)
  basket = dataset(b,:);
  [~,items_in_basket]=find(basket);
  for product=items_in_basket
    unique_counts(product) += 1;
  endfor
endfor
fprintf("Item supports: %sn", disp(unique_counts))
```

Figure 7.4: Performing counting of single items for the first pass of the Apriori algorithm.

of the individual items of C_1. Once we iterate through all eight transactions, we observe that the distinct items numbered from 1 to 6 are included in 5, 3, 3, 5, 1 and 3 transactions, meaning that we have $F_1 = \{1, 2, 3, 4, 6\}$.

2. Once we have F_1, the second phase begins by determining the candidates of frequent item pairs, i.e.,

   $$C_2 = \{\{1,2\},\{1,3\},\{1,4\},\{1,6\},\{2,3\},\{2,4\},\{2,6\},\{3,4\},\{3,6\},\{4,6\}\}.$$

   Another pass over the transactional dataset confirms us that these are solely item pairs $\{1,4\}$ and $\{1,6\}$ from C_2 that manage to have a support greater than or equal to our frequency threshold with four and three occurrences, respectively. This means, that we have $F_2 = \{\{1,4\},\{1,6\}\}$.

3. The third phase of the Apriori algorithm works by first determining C_3. According to the strategy introduced earlier in Section 7.2.1, we look for pairs of item sets in F_2 that only differ on their last item in their ordered set representations. This time we have a single pair of item pairs in F_2 and this pair happens to fulfil the necessity condition which makes their combination a worthy set to be included in C_3. As such, we have $C_3 = \{\{1,4,6\}\}$. Another pass over all the individual transactions of the transactional dataset reveals us that the support of the item set $\{1,4,6\}$ is two, which means that the single frequent candidate item triplet fails to be indeed frequent, resulting in $F_3 = \emptyset$.

4. Since F_3 was the empty set, we cannot form any frequent candidate item quadruples and the Apriori algorithm terminates.

Based on the previous steps, we can conclude that the set of frequent item sets from the sample dataset included in Table 7.3 is that of $F_1 \cup F_2 \cup F_3 = \{\{1\}, \{2\}, \{3\}, \{4\}, \{6\}, \{1,4\}, \{1,6\}\}$

7.3 *Park–Chen–Yu algorithm*

The **Park–Chen–Yu algorithm** (or PCY for short) can be regarded as an extension over the Apriori algorithm. This improvement builds upon the observation that during the different iterations of the Apriori algorithm, the main memory is loaded in a very unbalanced way.

That is, during the first iteration, we only need to deal with item singletons in C_1. Starting with the second iteration of the Apriori algorithm, things can go extremely resource intensive as the amount of memory it requires is $O(|F_1|^2)$, with the possibility that $|F_1| \approx |C_1|$ which can easily be at the scale of 10^5 or even beyond that. It would be really nice to make actual use of the unused excess memory we might have during the first phase of the Apriori algorithm in a way that would allow us to do somewhat less work during the upcoming phase(s).

Obviously, this additional work we perform has to be less than the amount of work we would do anyway in the second phase, otherwise we would gain nothing from our additional work. The solution is to randomly assign item pairs together and count the support of these *virtual item pairs*. We choose the number of virtual item pairs to be substantially lower than the number of potential candidate item pairs, meaning that the costs of this extra bookkeeping we perform simultaneously to the first phase is substantially cheaper than the resource needs for the entire second phase.

The way we create virtual item pairs (and item sets in more general) is via the usage of hash functions. A very simple form of deriving virtual item set identifiers to some item set $\mathcal{I}$ could be obtained by summing the integer identifiers to be found in the item set, and taking the modulus of the resulting sum with some integer small enough that we can reserve a counter for each possible outcome of the hash function employed. We could, for instance employ $h(\mathcal{I}) = \left(\sum_{i \in \mathcal{I}} i\right) \mod 5$, where $\mod$ denotes the modulus operator. This means that we would need to keep track of at most five additional counters, one for each virtual item set potentially formed by h. As an example, the above hash function would yield the virtual item set identifier 0 to the item set $\{4,5,6\}$.

? What potential pros and cons can you think of for applying multiple hash functions for creating virtual item sets?

The **pigeonhole principle** ensures that there need to be multiple actual item sets that get assigned to the same virtual identifier. Recall that for the previously proposed hash function, the item set $\{1,3,6\}$

would also yield the virtual item set identifier 0 which is identical for the item set $\{4, 5, 6\}$ as we have seen it before. This property of the virtual item set calculation further ensures that we can obtain a cheap upper bound on the supports of actual item sets by checking the support of the virtual item set an actual item set gets mapped to. This is because in practice we choose the number of virtual item sets to be orders of magnitude less than the number of potential item sets, which results in the fact that counters allocated for virtual item sets typically store the aggregated support of more than just one item set. Now, if the support belonging to the virtual item set – being mapped to multiple actual item sets – with identifier $h(\mathcal{I})$ happens to have a support below the minimum frequency threshold t, then there is absolutely no chance for any of the item sets that map to $h(\mathcal{I})$ to be frequent.

It is important to notice that the necessity condition purely based on the counts assigned to the virtual pairs of item sets does not subsume the necessity conditions of the standard Apriori algorithm. This means that – alongside the newly introduced counters for the virtual item pairs – we should still keep those counters that we used in the Apriori algorithm as well. If we do so, we can ensure that the number of candidates generated by PCY would never surpass the number of frequent item set candidates obtained by the Apriori algorithm.

7.3.1 *An example for the PCY algorithm*

Consider the same example transactional dataset from Table 7.3. Remember that the individual items numbered from 1 to 6 had supports being equal to $5, 3, 3, 5, 1$ and 3, respectively.

As mentioned in Section 7.3, the Park-Chen-Yu algorithm requires the introduction of virtual items sets that we can achieve via the application of hash functions. In this example, we use the hash function

$$h(x, y) = (5(x + y) - 3 \mod 7) + 1$$

in order to assign an actual item pair (x, y) into one of the 7 possible virtual item sets.

Table 7.4 contains in its upper triangular the virtual item set identifiers that an actual item set gets hashed to, when using them according to the above hash function. The lower triangular of Table 7.4 stores the actual support of item pairs. The lower triangular of the table is exhaustive in the sense that it contains supports for all possible item pairs. Recall that the Apriori and PCY algorithms exactly strive to actually quantify as few of these actual supports as possible by setting up efficiently computable necessity conditions that tell us if a pair of items has zero probability of being frequent.

It is hence important to emphasize that not all values of the lower triangular from Table 7.4 would get quantified during the execution of PCY, it simply serves the purpose of enumeration of all the pairwise actual occurrences of the item pairs. Notice that the support values in the main diagonal of Table 7.4 (those put in parenthesis) contain the support of single items that get calculated during the first phase of the algorithm.

	1	2	3	4	5	6
1	(5)	6	4	2	7	5
2	0	(3)	2	7	5	3
3	1	2	(3)	5	3	1
4	4	1	1	(5)	1	6
5	1	0	0	1	(1)	4
6	3	0	0	2	0	(3)

Table 7.4: The upper triangular of the table lists virtual item pair identifiers obtained by the hash function $h(x, y) = (5(x + y) - 3 \mod 7) + 1$ for the sample transactional dataset from Table 7.3. The lower triangular values are the actual supports of all the item pairs, with the values in the diagonal (in parenthesis) include supports for item singletons.

According to the values in the upper triangular of Table 7.4, whenever item pair $(1, 2)$ is observed in a transaction of the sample transactional dataset from Table 7.3, we increase the counter that we initialized for the virtual item pair with identifier 6. Table 7.4 tells us likewise that item pair $(1, 3)$ belongs to the virtual item pair 4. Figure 7.5 contains a sample implementation for obtaining all the (meta)supports obtained during the first pass of PCY.

As the PCY algorithm processes the entire transactional dataset, we end up seeing the item singleton and virtual item pair counters as listed in Table 7.5.

Item	1	2	3	4	5	6
Support	5	3	3	5	1	3

(a) Counters created for item singletons.

Virtual item pair	1	2	3	4	5	6	7
Support	1	6	0	1	4	2	2

(b) Counters for virtual item pairs.

Table 7.5: The values stored in the counters created during the first pass of the Park-Chen-Yu algorithm over the sample transactional dataset from Table 7.3.

From the counters included in Table 7.5, we can see that there were all together 20 items and 16 item pairs purchased in the sample input transactional dataset. Perhaps more importantly, it is also apparent from these counters that none of the item pairs involving item 5, nor those that hash to any of the virtual item pair identifiers $1, 3, 4, 6$ and 7 have a non-zero chance of being frequent in reality. Note that the first necessity condition that we know from the counters that we would also have in a vanilla Apriori implementation, however, the second criterion is originating from the application of PCY.

CODE SNIPPET

```
baskets=[1 1 1, 2 2 2, 3 3, 4 4 4, 5 5, 6 6, 7 7 7, 8 8];
items = [1 3 4, 1 4 5, 2 4, 1 4 6, 1 6, 2 3, 1 4 6, 2 3];
dataset = sparse(baskets, items, ones(size(items)));

unique_counts=zeros(1, columns(dataset));
num_of_virtual_baskets=7;
hash_fun=@(x,y) mod(5*(x+y)-3, num_of_virtual_baskets)+1;
virtual_supp=zeros(1, num_of_virtual_baskets);

for b=1:rows(dataset)
  basket = dataset(b,:);
  [~,items_in_basket]=find(basket);
  for product=items_in_basket
    unique_counts(product) += 1;
    for product2=items_in_basket
      if product<product2
        virtual_pair=hash_fun(product, product2);
        virtual_supp(virtual_pair) += 1;
      endif
    endfor
  endfor
endfor

fprintf("Item supports: %s\n", disp(unique_counts))
fprintf("Virtual item pair supports: %s\n", disp(virtual_supp))
```

Figure 7.5: Performing counting of single items for the first pass of the Park-Chen-Yu algorithm.

Recall that in the case of vanilla Apriori algorithm C_2 had $\binom{5}{2} = 10$ elements as it contained all the possible item pairs formed from the item set of frequent item singletons $F_1 = \{1, 2, 3, 4, 6\}$ with $|F_1| = 5$. In the case of PCY, however, we additionally know, that only those item pairs with virtual item pair identifier 2 or 5 have any chance of having a support of 3 or higher. There are only five item pairs that map to any of the frequent virtual item pairs, i.e., $\{1, 4\}, \{1, 6\}, \{2, 3\}, \{2, 5\}, \{3, 4\}$ (the virtual item identifiers written in red in Table 7.4).

Since all the necessity conditions have to hold at once for an item pair in order to be recognized as a potential frequent item pair, we can conclude that PCY identifies C_2 as $\{\{1, 4\}, \{1, 6\}, \{2, 3\}, \{3, 4\}\}$. Notice that this set is lacking item pair $\{2, 5\}$, which item pair would qualify it as a potentially frequent item pair based on its virtual item pair identifier, however, it fails to do so otherwise, as it contains an item singleton that is identified as being non-frequent (cf. item 5). This illustrates that the necessity conditions based on the virtual item sets are not strictly stronger, but play a complementary role to the necessity conditions imposed by the traditional Apriori algorithm.

By the end of the second pass over the transactional dataset, we obtain the counters to the item pairs of C_2 as included in Table 7.6. Based on the content of Table 7.6, we conclude that with our frequency threshold being defined as 3, we have $F_2 = \{\{1, 4\}, \{1, 6\}\}$.

Candidate item pair	$\{1, 4\}$	$\{1, 6\}$	$\{2, 3\}$	$\{3, 4\}$
Support	4	3	2	1

Table 7.6: Support values for C_2 calculated by the end of the second pass of PCY.

In the final iteration of PCY, we have $C_3 = \{\{1, 4, 6\}\}$. As our subsequent iteration over the transactional dataset would provide us with the information that $\{1, 4, 6\}$ has a support of 2, which means that for our frequency threshold of 3, we have $F_3 = \emptyset$. This additionally means that the PCY algorithm terminates, as we can now be certain – according to the Apriori principle – that none of the item quadruples have any chance of being frequent, that is we have $C_4 = \emptyset$.

7.4 *FP–Growth*

Apriori algorithm and its extensions are appealing as they are both easy to understand and implement in the form of an iterative algorithm. This iterative nature means that we do not need to keep the entire market basket dataset in the main memory, since it suffices to process one basket at a time and only allocate memory for the candidate frequent item sets and their respective counters.

We might, however, also face some drawbacks for applying the previously introduced algorithms for extracting frequent item sets as the repeated passes over the entire transactional database can be time-consuming. These repeated iterations can be especially time-consuming if it involves accessing the secondary memory from iteration to iteration.

Approaches capable of extracting frequent item sets without an iterative candidate set generation[1] could serve as a promising alternative to the previously discussed approaches that require multiple iterations over the transactional dataset.

[1] Han et al. 2000

The basic idea behind the **FP-Growth** algorithm is that the transactions in the market basket dataset can be efficiently compressed as the contents of the market baskets tend to overlap. The FP-Growth Algorithm achieves the above mentioned compression by relying on a special data structure named **frequent-pattern tree** (**FP-tree**). This special data structure is indented to store the entire transactional dataset and provides a convenient way to extract the support of item sets in an efficient manner.

The high-level working mechanism of the FP-Growth algorithm is the following:

1. we first build an FP-tree for efficiently storing the contents of the market baskets included in our transactional dataset,

2. we derive conditional datasets from the FP-tree containing the entire dataset and gradually expand it for finding frequent item sets of increasing cardinality in a recursive manner.

We face an obvious limitation of the above approach when our transactional data set is so large that even its compressed form in the FP-tree data structure is too large to fir in the main memory. One approach in such cases could be to randomly sample transactions from the transactional dataset and build the FP-tree based on that. If our sample is representative of the entire transactional dataset, then we shall experience similar relative supports for the different item sets we would experience otherwise if we relied on all the transactions.

7.4.1 *Building FP-trees*

Building an FP-tree goes by processing transactions from the transactional database one by one and creating and/or updating a path in the FP-tree dataset. In the beginning, we naturally start with an empty FP-tree.

FP-tree needs an ordering over the items such that the items within a basket shall be represented uniquely when we order the

items comprising the transaction. The ordering applied can be arbitrary, however, there is a typical one which sorts items based on their decreasing support. This useful heuristic usually helps the FP-tree data structure to obtain a better compression rate. The number of nodes assigned to an item in an FP-tree always ranges between one and the number of transactions the particular item is included in. It is also true, that items which are ranked higher in the ordering over the items would have fewer nodes assigned to them in the FP-tree.

The higher support an item has, the more problems it can cause when ranked low in our ordering as we are risking the introduction of as many nodes into the FP-tree that is equal to its support. It is hence a natural strategy to rank items with the largest support the highest, since this way we are more likely to avoid FP-trees with an increased number of internal nodes.

Once we have some ordering of the items, we can start processing the transactions of the transactional dataset sequentially. The goal is to include the contents of every basket from the transactional dataset in the FP-tree as a path starting at the root node. Every node in the FP-tree corresponds to an item and every node has an associated counter which indicates the number of transactions the item was involved. An edge between two nodes – corresponding to a pair of items – indicate that the corresponding items were located as adjacent items in at least one basket according to the ordered representations of the baskets.

For each transaction, we take the ordered items they are comprised of and update the FP-tree accordingly. The way an update looks is that we take the ordered items in a basked one by one and we see if the succeeding item from the basket can be found an a path in the FP-tree which starts at the root. If the succeeding item from the basket is already included at the FP-tree along our current path , then we only need to increase the associated counter for the node in the FP-tree which corresponds to our next item from the currently processed basket. Otherwise, we introduce a new node for the item in the FP-tree that we were not able to proceed to continuing the path that we started from the root of the FP-tree and initialize the associated counter of the newly created node to 1.

Since the same item can be part of multiple paths in the FP-tree, an item can be distributed over multiple nodes in the FP-tree. In order to support efficient aggregation of the same item – without the need of traversing the entire FP-tree – links between nodes referring to the same item are maintained in FP-trees. We can easily collect the total occurrence of an item by following these auxiliary links.

For illustrating the procedure of creating an FP-tree, let us consider the sample transactional dataset in Table 7.7. Figure 7.6 illus-

Transaction ID	Basket
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

Table 7.7: Example market basket database.

trates the inclusion of the first five market baskets from the example transactional dataset into an initially empty FP-tree. Starting with the insertion of the third market basket, we can see that the pointers connecting items of the same kind get introduced. These pointers are marked by red dashed edges in Figure 7.6 (c)–(e).

By looking at Figure 7.6 (e), we can conclude that the first five transactions of our transactional dataset contained item `E` twice, due to the fact that two is the sum for the counters associated to the nodes assigned to item `E` across the FP-tree. As including all these auxiliary edges to the FP-tree during the processing of the further transactions would result in a rather chaotic figure, we are omitting them in the followings.

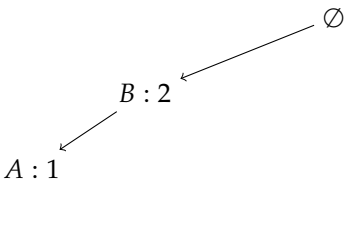
Figure 7.7 (a) presents us the FP-tree that we get after including all the transactions from the example transactional dataset included in Table 7.7. Let us emphasize that the auxiliary edges connecting the same item types play an integral role in FP-trees, we deliberately decided not to mark them for obtaining a more transparent figure.

Notice that the resulting FP-tree in Figure 7.7 (a) was based on the frequently used heuristic when items are ordered based on their decreasing order of support. Should we apply some other ordering of the items, we would get a differently arranged FP-tree. Indeed, Figure 7.7 (b) illustrates the case that we would get if the items were ordered based on their alphabetical ordering.

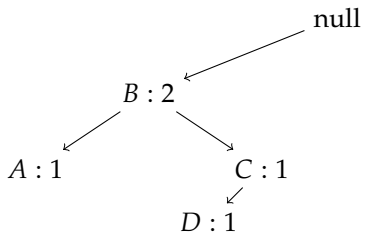
? What FP-trees would we get if items within baskets were ordered according to their increasing/decreasing order of support?

7.4.2 *Creating conditional datasets*

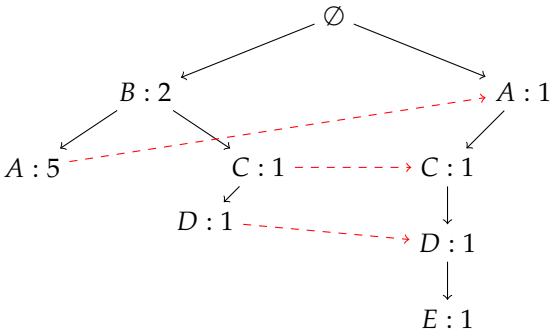
Once the entire FP-tree is build – presumably using the heuristic which orders the contents of transactions according to their decreasing support – we are ready to determine frequent item sets from the transactional dataset without any additional processing of the transactional dataset itself. That is, from that point on we are able to



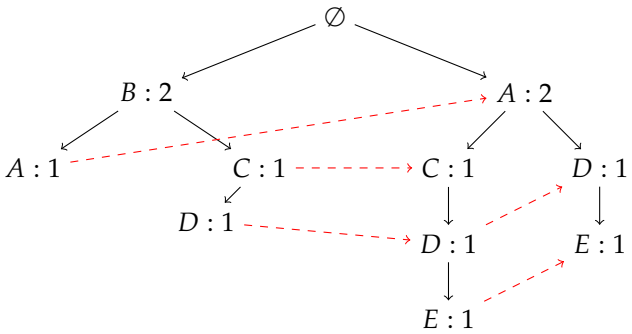
(a) After the 1st basket



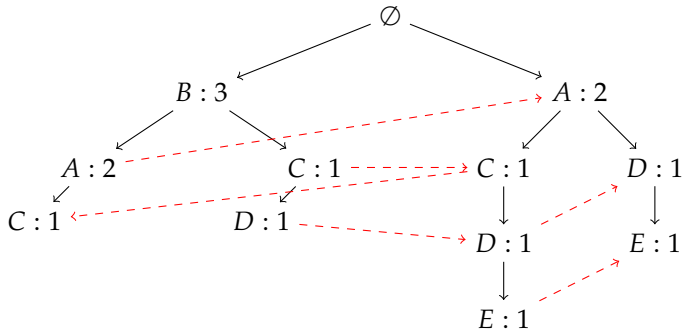
(b) After the 2nd basket



(c) After the 3rd basket

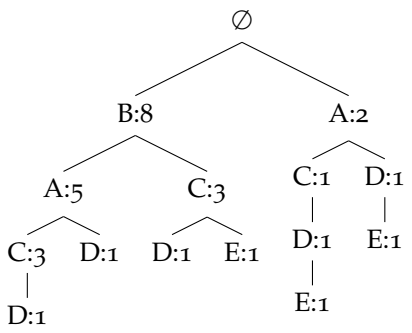


(d) After the 4th basket

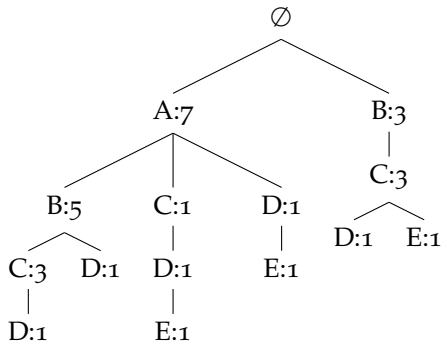


(e) After the 5th basket

Figure 7.6: The FP-tree over processing the first five baskets from the example transactional dataset from Table 7.7. The red dashed links illustrate the pointers connecting the same items for efficient aggregation.



(a) Using item ordering based on their support ($B > A > C > D > E$)



(b) Using alphabetical ordering of items ($A > B > C > D > E$)

Figure 7.7: FP-trees obtained after processing the entire sample transactional database from Table 7.7 when applying different item ordering. Notice that the pointers between the same items are not included for better visibility.

extract frequent item sets from the FP-tree directly and recursively.

The extraction of frequent item sets can be performed by using conditional FP-trees. **Conditional FP-trees** are efficiently determinable subtrees of an FP-tree which can help us to identify frequently co-occurring item sets. The benefit of these conditional FT-trees is that they can be directly determined from the FP-tree that we constructed earlier from the entire transactional dataset and there is no need for further multiple processing steps to be performed over the transactional dataset.

When we build an FP-tree conditionated over an item set $\mathcal{I}$, we are deriving such an FP-tree we would have constructed if the transactional dataset consisted from those transactions alone which contain item set $\mathcal{I}$. Table 7.8 illustrates the kind of dataset we would have if we discarded those transactions which are not a superset for the item set consisting of the single element `{E}`.

We could naturally rely on the entire transactional dataset and derive a reduced one which excludes those transactions that do not contain a certain item set. This would nonetheless be a rather inefficient strategy to choose. The important thing during the construction of a conditional FP-tree is that we never construct it from scratch, but derive them recursively from the unconditional transactional dataset that is build based on the entire dataset. Figure 7.8 depicts the condi-

TID	Basket
1	~~{A,B}~~
2	~~{B,C,D}~~
3	{A,C,D,E}
4	{A,D,E}
5	~~{A,B,C}~~
6	~~{A,B,C,D}~~
7	~~{B,C}~~
8	~~{A,B,C}~~
9	~~{A,B,D}~~
10	{B,C,E}

Table 7.8: The transactional dataset indicating irrelevant transactions with respect item set `{E}`. Irrelevant baskets are striked.

tional FP-trees we get based on the FP-trees from Figure 7.7.

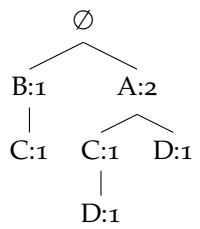
As a first step for creating a conditional FP-tree, we identify those nodes within the unconditional FP-tree that we would like to condition our FP-tree on. This step is as simple as following the pointers of the auxiliary links that connect the same items distributed across the FP-tree. All subtrees from the FP-tree that do not include any of the items we condition our data structure on can be pruned.

What we additionally need to do is adjusting the respective counters assigned to the individual nodes of the subtree. We can do it so in a bottom-up fashion by starting out at the items we are conditioning our FP-tree over and set the counters of the intermediate nodes of the conditioned FP-tree as the sum of the counters of those descendants of every node that we did not prune.

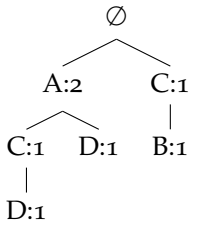
When conditioning the FP-trees from Figure 7.7 (a) and Figure 7.7 (b) on item set `{E}` in the above described manner, we get the conditional FP-trees included in Figure 7.8 (a) and Figure 7.8 (b), respectively.

From the conditional FP-trees obtained, we can easily identify those items that frequently co-occur with the item set we are currently conditioning our FP-tree on. All we need to do is to use the auxiliary links that tell us the number of times the item set we conditioned our FP-tree co-occurs with the further items. In the concrete example included in Figure 7.8 we can conclude that item set `{E}` co-occurs once with item `B` and twice with item `A`, `C` and `D`, respectively.

Assuming a frequency threshold of 2 would mean that item `E` does not form a frequent item pair with item `B`, however, item `E` forms a frequent item pair with all the remaining items, `A`, `C` and `D` with their joint support exactly being equal to our frequency threshold. We can now, build on top of the FP-tree that we just obtained conditioned on item set `{E}` and determine FP-trees conditioned on the expanded item set `{E,x}` with `x` being any of the items `A`, `C` or `D`.



(a) Using item ordering based on their support ($B > A > C > D > E$)



(b) Using alphabetical ordering of items ($A > B > C > D > E$)

Figure 7.8: FP-trees conditionated on item set `{E}` with different ordering of the items applied.

Since the procedure of creating a conditioned FP-tree can be naturally applied to an FP-tree which had already been obtained via conditioning on some item set, we can repeatedly apply the same procedure until we can find item sets that co-occur frequently enough. By continuing in an analogous manner, we can derive all the frequent item sets in our original transactional dataset without the need of iterating over it multiple times.

7.5 *Summary of the chapter*

This chapter introduced the task of frequent pattern mining from transactional datasets, where one is interested in the identification of those item which co-occur more frequently then some predefined threshold. The chapter additionally introduced the task of association rule mining, where our goal is to extract `if-then` type of rules from large datasets based on the frequent item sets identified.

This chapter also introduced the Apriori principle and two iterative algorithms, namely the eponymous Apriori and the Park-Chen-Yu (PCY) algorithms. These are iterative algorithms that gradually collect frequent item sets via a series of frequent candidate set generation and verification steps.

Related to these algorithms, this chapter also dealt with efficient ways of storing frequent item set candidates and their respective counters, as well as possible ways to perform candidate set generation.

At the end of the chapter, we introduced the FP-Growth algorithm. FP-Growth remarkably differs from the other algorithms discussed throughout this chapter in that it does not follow the principle of it-

erative candidate set generation. It instead builds on a special data structure, the frequent pattern trees (or FP-trees in short) which try to keep the entire transactional dataset in the main memory in a compressed and memory-efficient form. The FP-Growth algorithm extracts frequent item sets recursively via a divide-and-conquer mechanism through the construction of conditional FP-trees.

8 | Data mining from networks

Learning Objectives:
- Modeling by Markov Chains
- PageRank algorithm and its variants
- Hubs and Authorities

This chapter deals with the problem of analyzing complex networks or graphs. A graph is a mathematical object which – in their simplest forms – consist of vertices and edges. Graphs are powerful tools as they provide a natural representation for many real life concepts and phenomena, such as commonsense knowledge in the form of ontologies[1], taxonomies and knowledge graphs[2], social networks and interactions, and also more specialized ones including protein-protein interaction networks.

Readers of this chapter are expected to learn the working fundamentals related to different data mining algorithms applicable to complex networks. Additionally, readers should become proficient in assessing and interpreting the results obtained from such algorithms. A final aim of the chapter is to develop an understanding on the effects of the hyperparameters influencing the outcomes of the various algorithms and make them able to argue for particular choices of algorithms and hyperparameters.

[1] Miller 1995
[2] Speer et al. 2016

8.1 *Graphs as complex networks*

Complex networks are mathematical objects that are meant for describing real world phenomena and processes. These mathematical objects are often referred as graphs as well. A graph is a simple and highly relevant concept in graph theory and computer science in general. In its simplest form, a graph is a collection of nodes or vertices and a binary relation which holds for a subset of the pairs of vertices, which is indicated by edges connecting pairs of vertices within the graph for which the relation holds. As such graphs can be given as $G = (V, E)$, with V denoting its vertices and $E \subseteq V \times V$, with $\times$ referring to the Cartesian product of the vertices.

8.1.1 *Different types of graphs*

Complex networks can come in many forms, i.e., the edges can be both directed or undirected, weighted or unweighted, labeled or unlabeled. What directedness means in case of networks is that whenever an edge exist in one direction between two vertices, it is also warranted that the edge in the reverse direction can be found in the network, that is $(u, v) \in E \Leftrightarrow (v, u) \in E$. This happens when the underlying binary relation defined over the vertices is symmetric. For instance networks that represent which person – represented by a vertex – knows which other people in an institution would be best represented by an undirected graph. Ontological relations, such as being the subordinate of something, e.g. humans are vertebrates (but not vice versa), are on the other hand do not behave in a symmetric manner, hence the graph representing such knowledge would be undirected. Another example for undirected networks is the hyperlink structure of the world wide web, i.e., the fact that a certain website points do another one, does not imply that there also exists a hyperlink in the reverse direction.

When edges are weighted, it means that edges are not simply given in the form of $(u, v) \subseteq V \times V$, but as (u, v, w) which is a tuple representing not just a source and target node (u and v) but also some weight ($w \in \mathbf{R}$) that describes the strength of the connection between pairs of nodes.

Semantic networks, such as **WordNet** [3] and **ConceptNet**[4], are prototypical examples for labeled networks. Semantic networks have commonsense concepts as their vertices and there could kinds of relations hold between the vertices which are indicated by the labels of the edges. Taking the previous example, there is a directed edge labeled with the so-called Is-A relation between the vertex representing the concepts of *humans* and *vertebrates*. That is $(u, v, \text{Is-A}) \in E$, meaning that the Is-A relation holds for the concept pair (u, v) for the case when u and v are the vertices for concept of *humans* and *vertebrates*, respectively.

[3] Miller 1995

[4] Speer et al. 2016

As mentioned earlier, complex networks can be used to represent various processes and phenomena of every day life. Complex networks can be useful to model collaboration between entities, citation structure of scientific publications and various other kinds of social and economic interactions.

? Try to list additional uses cases when modeling a problem with networks can be applied.

8.1.2 *Representing networks in memory*

Networks of potential interest can range up to the point when they contain billions of vertices. Just think of the social network of Facebook for a very trivial example which had nearly 2.5 billion of active

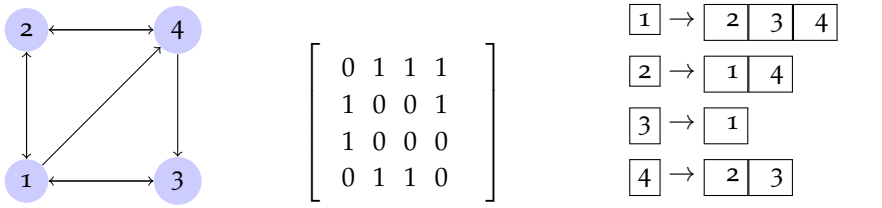
users over the first quarter of 2019 [5].

We should note that real-world networks are typically extremely sparse, i.e., $|E| \ll |V \times V|$, meaning that the vast majority of the potentially observed relations are not realized. In terms of a social network, even if the entire network has billions of nodes, the number of average connections per vertices is orders of magnitude smaller, say a few hundreds.

To this end, the networks are stored in a more efficient format, one of which is the **adjacency list** representation that is depicted in Figure 8.1 (c). The great benefit of the adjacency list representation is that it takes $O(|E|)$ amount of memory for storing the graph, as opposed to $O(|V^2|)$ which applies to the explicit **adjacency matrix** representation schematically displayed in Figure 8.1 (b) for the example directed graph from Figure 8.1 (a).

Similar to other situations, we pay some price for the efficiency of adjacency lists from the memory consumption point of view, as checking for the existence of an edge increases from $O(1)$ to $O(|V|)$ in the worst case scenario when applying an adjacency list instead of an adjacency matrix. This trade-off, however, is a worthy one in most real world situations when dealing with networks with a huge number of vertices and a relatively sparse link structure. Figure 8.2 illustrates how to store the example network from Figure 8.1 (a) in Octave when relying on both explicit dense and sparse representations.

[5] https://www.socialmediatoday.com/news/facebook-reaches-238-billion-users-beats-revenue-estimates-in-latest-upda/553403/



(a) Example digraph. (b) Adjacency matrix of the graph. (c) Adjacency list of the graph.

Figure 8.1: A sample directed graph with four vertices (a) and its potential representations as an adjacency matrix (b) and an edge list (c).

CODE SNIPPET

```
adjacency = [0 1 1 1; 1 0 0 1; 1 0 0 0; 0 1 1 0];

from_nodes = [1 1 1 2 2 3 4 4];
to_nodes = [2 3 4 1 4 1 2 3];
edge_weights = ones(size(to_nodes));
sparse_adjacency = sparse(from_nodes, to_nodes, edge_weights);
```

Figure 8.2: Creating a dense and a sparse representation for the example digraph from Figure 8.1.

8.2 Markov processes

Markov processes are often used to model real world events of sequential nature. For instance, a Markov model can be applied for modeling weather conditions or the outcome of sports events. In many cases, modeling sequential events is straightforward by determining a square matrix of transitional probabilities M, such that its entry m_{ij} quantifies the probability of observing the j^{th} possible outcome as our upcoming observation, given that we currently observe outcome i.

The main simplifying assumption in Markovian modeling is that the next upcoming thing we observe is not dependent on *all* our observations from the past, but it only depends upon our last (few) observations, i.e., we operate with a limited memory. It is like if we were making our weather forecast for the upcoming day purely based on the weather conditions we experience today. This is obviously a simplification in our modeling, but it also make sense not to condition our predictions for the future on events that had happened in the distant past.

To put it more formally, suppose we have some random variable X with possible outcomes indicated as $\{1, 2, \ldots, n\}$. By denoting our current observation towards random variable X at time t as X_t, what the Markovian assumption says is that

$$P(X_t = j | X_{t-1} = i, \ldots, X_0 = x_0) = P(X_t = j | X_{t-1} = i) = m_{ij}.$$

Naturally, since we would like to treat the elements of M as probabilities and every row of matrix M as a probability distribution, it has to hold that

$$\begin{aligned} m_{i,j} &\geq 0, \forall 1 \leq i, j \leq n, \\ \sum_{j=1}^{n} m_{i,j} &= 1, \forall 1 \leq i \leq n. \end{aligned} \tag{8.1}$$

Those matrices M that obey the properties described in 8.1 are called **row stochastic matrices**. As mentioned earlier, entry m_{ij} from such a matrix can be vied as the probability of observing the outcome of our random variable to be j immediately after an observation towards outcome i.

Example 8.1. *The next matrices qualify as row stochastic matrices:* $\begin{bmatrix} 0.1 & 0.9 \\ 0.2 & 0.8 \end{bmatrix}$, $\begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}$ *and* $\begin{bmatrix} 0.0 & 1.0 \\ 0.0 & 1.0 \end{bmatrix}$. *On the other hand, matrices* $\begin{bmatrix} -0.1 & 1.1 \\ 0.2 & 0.8 \end{bmatrix}$, $\begin{bmatrix} 0.2 & 0.9 \\ 0.2 & 0.8 \end{bmatrix}$ *and* $\begin{bmatrix} 0.8 & 0.7 \\ 0.2 & 0.3 \end{bmatrix}$ *do not count as row stochastic matrices (although the last one is a column stochastic matrix).*

MATH REVIEW | EIGENVALUES OF STOCHASTIC MATRICES

Matrices obeying the properties from 8.1 are called row stochastic matrices. A similar set of criteria can also be given for column stochastic matrices.

For stochastic matrices (let them be row or column stochastic matrices), we always have 1.0 as their largest eigenvalue. To see this simply look at the matrix-vector product obtained as $M\mathbf{1}$ for any row stochastic matrix M, with $\mathbf{1}$ denoting the vector consisting of all ones. Due to the fact that every row of M defines a probability distribution, we have that $M\mathbf{1} = \mathbf{1}$, meaning that 1 is always one of the eigenvalues of any (row) stochastic matrix.

It turns out that the previously seen eigenvalue of one is always going to be the largest among the eigenvalues of any stochastic matrix. In order to see why this is the case, let us define the `trace` matrix operator, which simply sums alls the elements of a matrix along its main diagonal. That is for some matrix M,

$$trace(M) = \sum_{i=1}^{n} m_{ii}.$$

From this previous definition, it follows that the trace of any n-by-n stochastic matrix is always smaller than or equal to n. Another property of the trace is that it can be expressed as the sum of the eigenvalues of the matrix. That is,

$$\sum_{i=1}^{n} \lambda_i = \sum_{i=1}^{n} m_{ii} = trace(M) < n.$$

What this means that in the 2-by-2 case – given that $\lambda_1 = 1, \lambda_2 < 1$ has to hold. From that point, we can use induction to see that for any n-by-n stochastic matrix we would have 1 as the largest (principal) eigenvalue.

Figure 8.3: Eigenvalues of stochastic matrices

8.2.1 *The stationary distribution of Markov chains*

A **stochastic vector** $\mathbf{p} \in \mathbf{R}^n_{\geq 0}$ is such a vector for which the sum of entries sums up to one, i.e., which forms a valid distribution. We can express a probabilistic belief over the different states of a Markov chain.

Notice that when we calculate $\mathbf{p}M$ for some stochastic vector $\mathbf{p}$ and row stochastic matrix M, we are essentially expressing the probability distribution over the different states of our Markov process for the next time step – relative to the actual time step based on our current beliefs for observing the individual states, as expressed by $\mathbf{p}$.

In order to make the dependence of our stochastic belief vector to the individual time step in the simulation, we will introduce $\mathbf{p}^{(t)}$ for making a reference to the stochastic vector from time step t. That is, $\mathbf{p}^{(t+1)} = \mathbf{p}^{(t)}M$. If we denote our initial configuration as $\mathbf{p}^{(0)}$, we can notice that $\mathbf{p}^{(t)}$ could just as well be expressed as $\mathbf{p}^{(0)}M^t$, because $\mathbf{p}^{(t)}$ is recursively entangled with $\mathbf{p}^{(0)}$.

It turns out, that – under mild assumptions that we shall discuss later – there exists a unique **steady state distribution**, also called as a **stationary distribution**, for which Markov chains described by a particular state transition matrix converge to. This distribution tells us the long term observation probability of the individual states of our Markov chain.

The stationary distribution for a Markov chain described by a row stochastic matrix M is the distribution $\mathbf{p}^*$ fulfilling the equation

$$\mathbf{p}^* = \lim_{t\to\infty} \mathbf{p}^{(0)}M^t.$$

In other words, $\mathbf{p}^*$ is the fixed point for the operation of right multiplication with the row stochastic matrix M. That is,

$$\mathbf{p}^* = \mathbf{p}^*M,$$

which implies that $\mathbf{p}^*$ is the left eigenvector of M corresponding to the eigenvalue 1. For row stochastic matrices, we can always be sure that eigenvalue 1 is among its eigenvalues and that this is going to be the largest eigenvalue (often called as the **principal eigenvalue**) for the given matrix.

Finding the eigenvector belonging to the principal eigenvalue is as simply as choosing an initial stochastic vector $\mathbf{p}$ and keep it multiplied by the row stochastic matrix M until there is no substantial change in the resulting vector. This simple procedure is named the **power method** and it is illustrated in Algorithm 3. The algorithm is guaranteed to converge to the same principal eigenvector no matter how we choose our initial stochastic vector $\mathbf{p}$.

Require: Input matrix $M \in \mathbf{R}^{n\times n}$ and a tolerance threshold ϵ
Ensure: principal eigenvector $\mathbf{p} \in \mathbf{R}^n$

```
function PowerIteration(M, ε)
    p = [1/n]_{i=1}^{n}        // an n-element vector consisting of all 1/n
    while ||p − pM||_2 > ε do
        p = pM
    end while
    return p
end function
```

Algorithm 3: The power algorithm for determining the principal eigenvector of matrix M.

8.2.2 *Markov processes and random walks — through a concrete example*

For a concrete example, let us assume that we would like to model the weather conditions of a town by a Markov chain. The three different weather conditions we can make are Sunny (observation #1, abbreviated as S), Overcast (observation #2, abbreviated as O) and Rainy (observation #3, abbreviated as R).

Hence, we have $n = 3$, i.e., the number of different observations for the random variable describing the weather of the town has three possible outcomes. What it further implies that the Markov chain can be described as a 3-by-3 row stochastic matrix M, which tells us the probability for observing a pair of weather conditions for two consecutive days in all possible combinations (that is sunny-sunny, sunny-overcast, sunny-rainy, overcast-sunny, overcast-overcast, overcast-rainy, rainy-sunny, rainy-overcast, rainy-rainy).

The transition probabilities needed for matrix M can be simply obtained by taking the maximum likelihood estimate (MLE) for any pair of observations. All we need to do to obtain MLE estimates for transition probabilities is that we keep a long track of the weather conditions and simply rely on the definition of the conditional probability stating that

$$P(A|B) = \frac{P(A,B)}{P(B)}.$$

The definition of conditional probability employed for the determination of transition probabilities of our Markov chain simply means that for a certain pair of weather combination (w_i, w_j) in order to quantify the probability $P(w_j|w_i)$ what we have to do is to count the number of times we observed that a day with weather condition w_i was followed by weather condition w_j and divide that by the number of days weather condition w_i was observed (irrespective of the weather condition for the upcoming day). Suppose we have observed the weather to be sunny 5,000 times and that we have seen the weather after a sunny day to be overcast or rainy 500-500 times each. This also implies that for the remaining 5,000-500-500=4,000 cases, a sunny day was followed by another sunny day. In such a case, we would infer the following transition probabilities:

$$\begin{aligned} P(sunny|sunny) &= \frac{4000}{5000} = 0.8 \\ P(overcast|sunny) &= \frac{500}{5000} = 0.1 \\ P(rainy|sunny) &= \frac{500}{5000} = 0.1. \end{aligned}$$

In a likewise manner, one could collect all the MLE estimates for the transition probabilities for a Markov chain. Suppose we did so,

and obtained the matrix of transition probabilities between the three possible states of our Markov chain as

$$M = \begin{bmatrix} 0.8 & 0.1 & 0.1 \\ 0.2 & 0.4 & 0.4 \\ 0.2 & 0.1 & 0.7 \end{bmatrix}. \tag{8.2}$$

A state transition matrix, such as the one above, can be also imagined graphically. Figure 8.4 provides such a network visualization of our Markov chain. In the network interpretation of a Markov chain, each state of our random process corresponds to a node in the network and state transition probabilities correspond to the edge weights that tell us the probability for traversing from a certain start node to a particular neighboring node. The row stochasticity of the transition matrix in the network view can be seen as the sum of the weight for the outgoing edges of a particular node always sum to one.

We can now rely on this graphical representation of the state transition probability matrix as a way for modeling stochastic processes that are described as a Markov process in the form of random walks. A **random walk** describes a stochastic temporal process in which one picks a state of the Markov process sequentially and stochastically. That is, when the random walk is at a given state, it chooses its next state randomly, proportional to the state transition probabilities described in the matrix of transition probabilities. The stationary distribution in that view can be interpreted as fraction of time a random walk spends at each state of the Markov chain.

In our concrete example – where the weather condition is described by a Markov process with three states – we have $n = 3$. A stochastic vector $\mathbf{p} = [1, 0, 0]$ can be interpreted as observing weather condition #1 with absolute certainty, i.e., a sunny weather. Vector $\mathbf{p} = [0.5, 0.14, 0.36]$ on the other hand describes a stochastic weather configuration, where the sunny weather is the most likely outcome (50%), followed by a rainy weather (36%) and the overcast weather condition being regarded the least likely (14%).

In Figure 8.4, the radius of the nodes – each marking one of the states from the Markov chain – is drawn proportionally to the stationary distribution of the various states. Figure 8.5 contains a sample code for approximating the principal eigenvector of our state transition probability matrix M. The output of the sample code also illustrates the **global convergence** of the stationary distribution, since the different initializations for $\mathbf{p}^*$ (the rows of matrix P) all converged to a very similar solution already in 10 iterations. If the number of iterations was increased, the resulting vectors would resemble each other even more in the end.

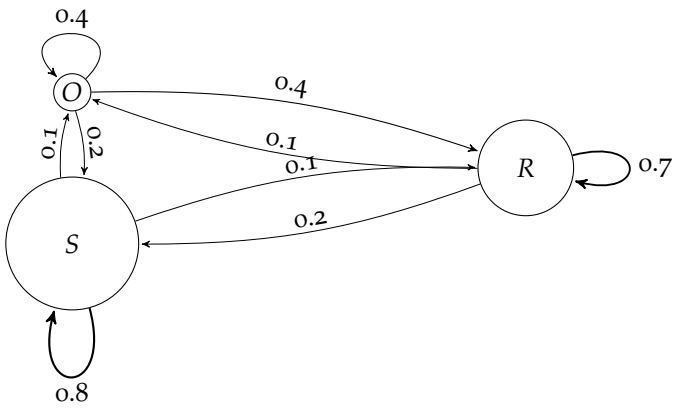


Figure 8.4: An illustration of the Markov chain given by the transition probabilities from 8.2. The radii of the nodes – corresponding to the different states of our Markov chain – if proportional to their stationary distribution.

In our concrete case, way we should interpret the stationary distribution $\mathbf{p}^* = [0.5, 0.14, 0.36]$ such that we would expect to see half of the days to be sunny and that there are approximately 2.5 times as many rainy days compared to overcast days *on the long run* given that our state transition probabilities accurately approximate the conditional probabilities between the weather conditions of two consecutive days.

CODE SNIPPET

```
M = [0.8 0.1 0.1;
     0.2 0.4 0.4;
     0.2 0.1 0.7];  % the state transition probability matrix

rand("seed", 42)    % fixing the random seed
P = rand(5, 3);     % generate 5 random vectors from [0;1]
P = P ./ sum(P, 2); % turn rows into probability distributions

for k=1:15
  P = P*M;          % power iteration for 15 steps
endfor
disp(P)

>> 0.49984   0.14286   0.35731
   0.49984   0.14286   0.35731
   0.49991   0.14286   0.35723
   0.49996   0.14286   0.35718
   0.49995   0.14286   0.35719
```

Figure 8.5: The uniqueness of the stationary distribution and its global convergence property is illustrated by the fact that all 5 random initializations for $\mathbf{p}$ converged to a very similar distribution.

8.3 PageRank algorithm

PageRank[6] is admittedly one of the most popular algorithms for determining importance scores for the vertices of complex networks. The impact of the algorithm is illustrated by the fact that it was given the Test of Time Award at the 2015 World Wide Web Conference, one of the leading conferences in computer science. The algorithm itself can be viewed as an instance of a Markov random process defined over the nodes of a complex network as the possible state space for the problem.

[6] Page et al. 1998

Although PageRank was originally introduced in order to provide an efficient ranking mechanism for webpages, it is useful to know that PageRank has inspired a massive amount of research that reach beyond that application. TextRank [7], for instance, is one of the many prototypical works that build on top of the idea of PageRank by using a similar working mechanism to PageRank for determining important concepts, called keyphrases from textual documents.

[7] Mihalcea and Tarau 2004

8.3.1 About information retrieval

As mentioned before, PageRank was originally introduced with a motivation to provide a ranking for documents on the web. This task fits into the broader task of **information retrieval** [8] applications.

[8] Manning et al. 2008

In information retrieval (IR for short), one is given with – a potentially large – collection of documents, often named a **corpus**. The goal of IR is to find and rank those documents from the corpus that are relevant towards a particular search need expressed in the form of a search query. Probably the most prototypical applications of information retrieval are search engines on the world wide web, where relevant documents have to be returned and ranked over hundreds of billions of websites.

Information retrieval is a complex task which involves many other components beyond employing some PageRank-style algorithm for ranking relevant documents. Information retrieval systems also have to handle efficient **indexing**, meaning that these systems should be able to return that subset of documents which contain some query expression. Queries could be multi-word units, and they are more and more often expressed as natural language questions these days. That is, people would rather search something along the lines of "*What is the longest river in Europe?*" instead of simply typing in expressions like "*longest European river*". As such, indexing poses several interesting questions, however, those are beyond the scope of our discussion.

Even in the case when people search for actual terms or phrases,

further restrictions might apply, such as returning only documents that has an exact match towards a search query, or exactly to the contrary, behaving tolerantly towards misspelling or grammatical inflections (i.e.,the word *took* is the past tense of the verb *take*). Additionally, indexing should also provide support for applying different boolean operators, such as the logical `OR`, `NOT` operators.

As the previous examples suggests, document indexing includes several challenging problems. Besides these previously mentioned problems, efficient indexing of large document collections is a challenging engineering problem in itself, since it is of utmost importance to return the set of relevant documents for a search query against document collections that potentially contain hundreds of millions of indexed words in milliseconds. A search engine should not only be fast in respect of individual queries, but it should also be highly concurrent, i.e., it is important that they can cope with vastly parallel usage.

Luckily, there exists platforms that one can rely when in need for a highly scalable and efficient indexing. We will not go into the details of these platforms, however, we mention two popular such frameworks, Solr[9] and Elasticsearch[10]. In what comes, we would assume that we have access to a powerful indexing service and we shall focus on the task of assigning an importance score for the vertices of either an entire network or a subnetwork which contains those vertices of a larger structure that already fulfil certain requirements, such as containing a given search query.

[9] https://lucene.apache.org/solr/

[10] https://www.elastic.co/elasticsearch/

8.3.2 *The working mechanisms of PageRank*

As mentioned earlier, PageRank can be viewed as an instance of a Markov random process defined over the vertices of a complex network as the possible state space for the problem. As such the PageRank algorithm can be viewed as a random walk model which operates over the different states of a Markov chain. When identifying the websites on the internet as the states of the Markov process, this random walk and the stationary network the random walk converges to has a very intuitive meaning.

Imagine a random surfer on the internet which starts off at some website and randomly picks the next site to visit from the websites that are directly hyperlinked from the currently visited site. Now, if we imagine that this random process continues long enough (actually infinitely long in the limit) and we keep track of the amount of time spent by the random walk at the individual websites, we get the stationary distribution of our network. Remember that the stationary distribution is insensitive to our start configuration, i.e., we shall

converge to the same distribution, no matter which was the initial website to start random navigation from. The probabilities in the stationary distribution can then be interpreted as the relevance of each vertex (websites in the browsing example) of the network. As such, when we would like to rank the websites based on their relevance, we could just sort them according to their PageRank scores.

The PageRank algorithm assumes that every vertex $v_j \in V$ in some network $G = (V, E)$ has some relevance score p_j and its relevance depends on the relevance scores of all its neighboring vertices $\{v_i | (v_i, v_j) \in E\}$. What the PageRank model basically says is that every vertex in the network distributes its importance towards all its neighboring vertices in an even manner and that the importance of a vertex can be determined as the sum of the importances transferred from its neighbors via an incoming edge. Formally, the importance score of vertex v_j is determined as

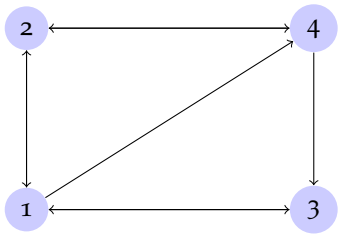
$$\mathbf{p}_j = \sum_{(i,j)\in E} \frac{1}{d_i} \mathbf{p}_i, \tag{8.3}$$

where d_i denotes the out-degree of vertex v_i.

Essentially, when calculating the importance of the nodes according to the PageRank model, we are calculating the stationary distribution of a Markov chain with a special state transition matrix M which either assumes that the probability of transition between two states is zero (when there is no direct hyperlink between a pair of websites) or it is uniformly equal to $\frac{1}{d_i}$.

It is easy to see that the above mentioned strategy for constructing the state transition matrix M results in M being a row stochastic matrix, i.e., all of its rows will contain nonnegative values and sum up to one, assuming that every vertex has at least one outgoing edge. We shall soon provide a remedy for the situation when certain nodes have no outgoing edges.

? What makes this choice of state transition probabilities counterintuitive? Can you nonetheless argue for this strategy? (Hint: think of the principle of maximum entropy.)



(a) The example network.

$$\begin{bmatrix} 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix}$$

(b) The corresponding state transition matrix

Figure 8.6: An example network and its corresponding state transition matrix.

Example 8.2. *Figure 8.6 contains a sample network and the state transition matrix which describes the probability of transitioning between any pair of websites. Table 8.1 illustrates the convergence of the power iteration over*

the state transition matrix M of the sample network from Figure 8.6. We can conclude that vertex 1 has the highest prominence within the network (cf. $\mathbf{p}_1^* = 0.333$), and all the remaining vertices have an equal amount of importance within the network (cf. $\mathbf{p}_j^* = 0.222$ for $j \in \{2,3,4\}$).

$\mathbf{p}^{(0)}$	$\mathbf{p}^{(1)}$	$\mathbf{p}^{(2)}$	$\mathbf{p}^{(3)}$	...	$\mathbf{p}^{(6)}$	...	$\mathbf{p}^{(9)}$
0.25	0.375	0.313	0.344	...	0.332	...	0.333
0.25	0.208	0.229	0.219	...	0.224	...	0.222
0.25	0.208	0.229	0.219	...	0.224	...	0.222
0.25	0.208	0.229	0.219	...	0.224	...	0.222

Table 8.1: The convergence of the PageRank values for the example network from Figure 8.6. Each row corresponds to a vertex from the network.

In our example network from Figure 8.6, the $\mathbf{p}_2^* = \mathbf{p}_3^*$ relation naturally holds as vertex 2 and 3 share the same neighborhood regarding their incoming edges. But why is it tha case, that $\mathbf{p}_4^*$ also has the same value?

In order to see that, simply write up the definition for the stationary distribution for vertex 1 by its recursive formula, i.e.,

$$\mathbf{p}_1^* = \frac{1}{2}\mathbf{p}_2^* + \mathbf{p}_3^*.$$

Since we have argued that $\mathbf{p}_2^* = \mathbf{p}_3^*$, we can equivalently express the previous equation as

$$\mathbf{p}_1^* = \frac{3}{2}\mathbf{p}_2^* = \frac{3}{2}\mathbf{p}_3^*.$$

If we now write up the recursive formula for the PageRank value of vertex 4, we get

$$\mathbf{p}_4^* = \frac{1}{3}\mathbf{p}_1^* + \frac{1}{2}\mathbf{p}_2^* = \frac{1}{3}\frac{3}{2}\mathbf{p}_2^* + \frac{1}{2}\mathbf{p}_2^* = \mathbf{p}_2^*.$$

8.3.3 *The ergodicity of Markov chains*

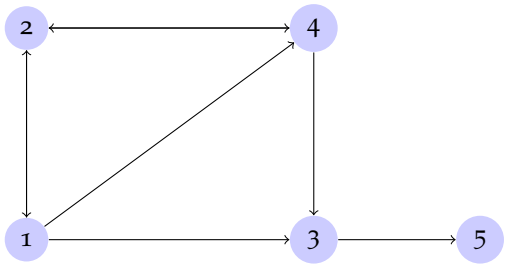
We mentioned earlier that we obtain a useful stationary distribution our random walk converges to if our Markov chain has a nice property. We now introduce the circumstances when we say that our Markov chain behaves nicely and this property is called **ergodicity**. A Markov chain is *ergodic* if it is *irreducible* and *aperiodic*.

A Markov chain has the **irreducibility** property, if there exists at least one path between any pair of states in the network. In other words, there should be a non-zero probability for transitioning between any pairs of states. The example Markov chain visualized in Figure 8.6 had this property.

A Markov chain is called aperiodic if all of its states are aperiodic. The **aperiodicity** of a state means that whenever we start out from that state, we do not know the exact number of steps when we would

return next to the same state. The opposite of aperiodicity is periodicity. In the case a state is periodic, there is a fixed number k such that we know that we would return to the same state in every k step.

Although the Markov chains depicted in Figure 8.7 and Figure 8.8 are very similar to the one in Figure 8.6, they differ in small but important ways. Figure 8.6 includes an ergodic Markov chain, whereas the ones in Figure 8.7 and Figure 8.8 are not an irreducible or an aperiodic one. In the followings, we shall review the problems that arise with the stationary distribution of such non-ergodic networks.



(a) Network with a dead-end (cf. node 5) causing non-irreducibility.

$$\begin{bmatrix} 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(b) The corresponding state transition matrix.

Figure 8.7: An example network which is not irreducible with its state transition matrix. The problematic part of the state transition matrix is in red.

The problem of dead ends Irreducibility requires that there exist a path between any pair of states in the Markov chain. In the case of the Markov chain included in Figure 8.7 this property is clearly violated, as state 5 has no outgoing edges. That is not only there is no path to all states from state 5, but there is not a single path to *any* of the states starting from state 5. This is illustrated by a row in the state transition matrix with all zeros in Figure 8.7 (b).

Such nodes from a random walk perspective means that there is a certain point such that we cannot continue our walk as there is no direct connection to proceed towards. As the stationary distribution is a modeling that is performed in the limit on infinite time horizon, it is intuitively inevitable that sooner or later the random walk gets into this **dead end** situation from which there is no further way to continue the walk.

In order to see more formally, why nodes with no outgoing edges cause a problem, consider a small Markov chain with a state transition matrix

$$M = \begin{bmatrix} \alpha & 1-\alpha \\ 0 & 0 \end{bmatrix}, \tag{8.4}$$

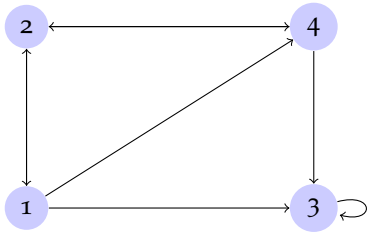
for some $1 > \alpha > 0$. Remember that the stationary distribution of a Markov chain described by transition matrix M is

$$\mathbf{p}^* = \lim_{t \to \infty} \mathbf{p}^{(0)} M^t. \tag{8.5}$$

In the case when M takes on the form given in 8.4, we have that

$$\lim_{t\to\infty} M^t = \lim_{t\to\infty} \begin{bmatrix} \alpha^t & (1-\alpha)^t \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix},$$

which means that when multiplied by any $\mathbf{p}^{(0)}$ in 8.5, we would get the zero vector as our stationary distribution. What this intuitively means, is that in a Markov chain with a dead end, we would eventually and inevitably get to the point where we could not continue our random walk on an infinite time horizon. Naturally, importance scores that are all zeros for all the states are meaningless, hence we would need to find some solution to overcome the problem of dead ends in Markov chains.



(a) Network with a cycle (cf. node 3) causing non-aperiodicity.

$$\begin{bmatrix} 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix}$$

(b) The corresponding state transition matrix.

Figure 8.8: An example network which is not aperiodic with its state transition matrix. The problematic part of the state transition matrix is in red.

The problem of spider traps As the other problem source, let us focus on spider traps. **Spider traps** arise when the aperiodicity of the network is violated. We can see the simplest form of violation of aperiodicity in the form of a state with no other outgoing edges beyond a self-loop. An example for that being the case can be seen in Figure 8.8. Note that in general it is possible that a Markov chain contains "coalitions" of more than just a single state. Additionally, multiple such problematic sub-networks could co-exist within a network simultaneously.

Table 8.2 illustrates the problem caused by the periodic nature of state 3 regarding the stationary distribution of the Markov chain. We can see that this time the probability mass accumulates within the subnetwork which forms the spider trap. Since the spider trap was formed by state 3 alone, the stationary distribution of the Markov chain is going to be such a distribution which is 1.0 for state 3 and zero for the rest of the states.

Such a stationary distribution is clearly just as useless as the one that we saw for the network structure that contained a dead end. We shall devise a solution which mitigates the problem caused by possible dead ends and spider traps in networks.

? What interpretation would you give to the resulting stationary distribution? (Hint: look back at the interpretation we gave for the resulting stationary distribution for Markov chains with dead ends.)

$\mathbf{p}^{(0)}$	$\mathbf{p}^{(1)}$	$\mathbf{p}^{(2)}$	$\mathbf{p}^{(3)}$	...	$\mathbf{p}^{(6)}$	...	$\mathbf{p}^{(9)}$
0.25	0.125	0.104	0.073	...	0.029	...	0.011
0.25	0.208	0.146	0.108	...	0.042	...	0.016
0.25	0.458	0.604	0.712	...	0.888	...	0.957
0.25	0.208	0.146	0.108	...	0.042	...	0.016

Table 8.2: Illustration of the power iteration for the periodic network from Figure 8.8. The cycle (consisting of a single vertex) accumulates all the importance within the network.

8.3.4 *PageRank with restarts — a remedy to dead ends and spider traps*

As noted earlier, the non-ergodicity of a Markov chain causes the stationary distribution towards some degenerate distribution that we cannot utilize for ranking its sates. The way the PageRank algorithm overcomes the problems arising when dealing with non-ergodic Markov chains via the introduction of a **damping factor** β. This *beta* coefficient is employed in a way that the original recursive connection between the importance score of a vertex and its neighbors as introduced in 8.3 changes to

$$\mathbf{p}_j = \frac{1-\beta}{|V|} + \sum_{(i,j)\in E} \frac{\beta}{d_i}\mathbf{p}_i, \tag{8.6}$$

with d_i denoting the number of outgoing edges from vertex i. The above formula suggests that the damping factor acts as a discounting factor, i.e., every vertex redistributes β fraction of its own importance towards its direct neighbors. As a consequence, $1-\beta$ probability mass is kept back, which can be evenly distributed across al the vertices. This is illustrated by the $\frac{1-\beta}{|V|}$ term in 8.6.

An alternative way to look at the damping factor is that we are performing an interpolation between two random walks, i.e., one that is based on our original Markov chain with probability β and another Markov chain which has the same state space, but which has a fully connected state transition structure with all the probabilities being set to $\frac{1-\beta}{|V|}$.

From the random walk point of view, this can be interpreted as choosing a link to follow from our original network with probability β and performing a hop to an randomly chosen vertex with probability $1-\beta$. The latter can be viewed as restarting our random walk occasionally. The choice for β affect how frequently does our random walk gets restarted in our simulation, which implicitly affects the ability of our random walk model to handle the potential non-ergodicity of our Markov chain.

We next analyze the effects of choosing β, the value for which is typically set to a value moderately smaller than 1.0, i.e., around 0.8 and 0.9. The probability of restarting the random walk as a function of the damping factor β follows a **geometric distribution** with success probability $1-\beta$.

Table 8.3 demonstrates how different choices for β affect the probability of restarting the random walk as different number of consecutive steps are performed in the random walk process. The values in Table 8.3 convey the reassuring message that the probability of performing a restart asymptotes to 1 in an exponential rate, meaning that we manage to quickly escape from dead ends and spider traps in our network.

	$\beta = 0.8$	$\beta = 0.9$	$\beta = 0.95$
probability of a restart in 1 step	0.20	0.10	0.05
probability of a restart in 5 step	0.67	0.41	0.23
probability of a restart in 10 step	0.89	0.65	0.40
probability of a restart in 20 step	0.99	0.88	0.64
probability of a restart in 50 step	1.0	0.99	0.92

Table 8.3: The probability of restarting a random walk after varying number of steps and damping factor β.

Table 8.4 illustrates the beneficial effects of applying a damping factor as introduced in 8.6. Table 8.4 includes the convergence towards the stationary distribution for the Markov chain depicted in Figure 8.8 when using $\beta = 0.8$. Recall that the Markov chain from Figure 8.8 violated aperiodicity, which resulted in that all the probability mass in the stationary distribution accumulated for state 3 (the node which had a single-self loop as an outgoing edge). The stationary distribution for the case when no teleportation was employed – or alternatively, the extreme damping factor of $\beta = 1.0$ was employed –is displayed in Table 8.2. The comparison of the distributions in Table 8.2 and Table 8.4 illustrate that applying a damping factor $\beta < 1$ indeed solved the problem of the Markov chain being periodic.

$\mathbf{p}^{(0)}$	$\mathbf{p}^{(1)}$	$\mathbf{p}^{(2)}$	$\mathbf{p}^{(3)}$	...	$\mathbf{p}^{(6)}$	...	$\mathbf{p}^{(9)}$
0.25	0.150	0.137	0.121	...	0.105	...	0.101
0.25	0.217	0.177	0.157	...	0.134	...	0.130
0.25	0.417	0.510	0.565	...	0.627	...	0.639
0.25	0.217	0.177	0.157	...	0.134	...	0.130

Table 8.4: Illustration of the power iteration for the periodic network from Figure 8.8 when a damping factor $\beta = 0.8$ is applied.

8.3.5 *Personalized PageRank — random walks with biased restarts*

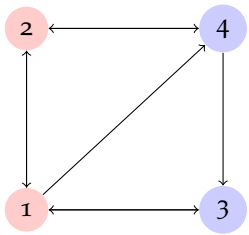
An alternative way to look at the damping factor β as the value which controls the amount of probability mass that vertices are allowed to redistribute. This also means that $1 - \beta$ fraction of their prestige is not transferred towards their directly accessible neighbors. We can think of this $1 - \beta$ fraction of the total relevance as a form of tax that we collect from the vertices. Earlier we used that amount

of probability mass to be evenly redistributed over the states of the Markov chain (cf. the $\frac{1}{|V|}$ term in 8.6).

There could, however, be such applications when we would like to prevent certain states from receiving a share from the probability mass collected for redistribution. Imagine that the states of our Markov chains correspond to web pages and we have the information that certain subset of the vertices belong to the set of trustful sources. Under this circumstance, we could bias our random walk model to only favor those states that we choose to based on their trustworthiness. To view this through the lense of restarts, what this means is that if we decide to restart our random walk then our new starting point should be one of the favored states that we trust. This variant of PageRank is named the **TrustRank** algorithm.

Another meta-information which could serve as the basis of restarting a random walk over the states corresponding to websites could be based on their topical categorization. For instance we might have a classification of the websites available whether their topic is about sports or not. When we would like to rank websites for a user with a certain topical interest then it could be a good idea to determine the PageRank scores by such a random walk model which performs restarts favoring those states that are classified in accordance with the topical interest(s) of the given user. This variant of PageRank which favors certain subset of states upon restart based on user preferences is called the **Personalized PageRank** algorithm. In the most extreme case, the subset of states that we would like to favor might be a single state.

Example 8.3. *Suppose we deal with the very same network structure as already presented in Figure 8.6. When calculating the PageRank scores, we apply a damping factor* $\beta = 0.8$ *and we regard states* 1 *and* 2 *as the only states where random walks can (re)start. This is illustrated in Figure 8.9.*



(a) Network with a cycle (cf. node 3) causing non-aperiodicity.

$$\begin{bmatrix} 0 & \frac{\beta}{3} & \frac{\beta}{3} & \frac{\beta}{3} \\ \frac{\beta}{2} & 0 & 0 & \frac{\beta}{2} \\ \beta & 0 & 0 & 0 \\ 0 & \frac{\beta}{2} & \frac{\beta}{2} & 0 \end{bmatrix}$$

(b) The corresponding state transition matrix when applying a damping factor of 0.8.

$$\begin{bmatrix} \frac{1-\beta}{2} & \frac{1-\beta}{2} & 0 & 0 \\ \frac{1-\beta}{2} & \frac{1-\beta}{2} & 0 & 0 \\ \frac{1-\beta}{2} & \frac{1-\beta}{2} & 0 & 0 \\ \frac{1-\beta}{2} & \frac{1-\beta}{2} & 0 & 0 \end{bmatrix}$$

(c) Auxiliary transition matrix for the biased random walk.

Figure 8.9: An example (non-ergodic) network in which we perform restarts favoring state 1 and 2 (indicated by red background).

We can notice that the stationary distribution for the random walk implemented over the Markov chain in Figure 8.9 can also be re-

garded as the stationary distribution of the sum of the state transition matrices from Figure 8.9 (b) and Figure 8.9 (c). That is, the personalized PageRank scores that we obtain over the Markov chain from Figure 8.9 is the principal eigenvector for the matrix

$$0.8\begin{bmatrix} 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix} + 0.2\begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{10} & \frac{11}{30} & \frac{4}{15} & \frac{4}{15} \\ \frac{1}{2} & \frac{1}{10} & 0 & \frac{4}{10} \\ \frac{9}{10} & \frac{1}{10} & 0 & 0 \\ \frac{1}{10} & \frac{1}{2} & \frac{4}{10} & 0 \end{bmatrix}.$$

While it is certainly a valid way to look at the problem of PageRank with (biased) restarts as the stationary distribution of the interpolation of two state transition matrices, it is important to mention that in efficient implementations, we seldom calculate the interpolated state transition matrix. Instead of calculating the explicit interpolated state transition matrix, the typical solution is to repeatedly calculate $\mathbf{p}^{t+1} = \beta\mathbf{p}^{t+1}M$ and correct for the possibility of restarts by adding an appropriately chosen vector to the resulting product, i.e., $\mathbf{p}^{t+1} = \mathbf{p}^{t+1} + \frac{1-\beta}{|V|}\mathbf{1}$, where $\mathbf{1}$ denotes the vector of all ones.

Remember that the stationary distribution we obtained earlier for the Markov chain without applying a damping factor was

$$[0.333, 0.222, 0.222, 0.222]$$

as also included in Table 8.1. This time, however, when applying $\beta = 0.8$ and allowing restarts from state 1 and state 2 alone, we obtain a stationary distribution of

$$[0.349, 0.274, 0.174, 0.203].$$

As a consequence of biasing the random walk towards states $\{1, 2\}$, we observe an increase in the stationary distribution for these states. Naturally, as the amount of probability mass which moved to the favored states is now missing in total from the remaining two states. Even though originally $\mathbf{p}_3^* = \mathbf{p}_4^*$ held, we no longer see this in the stationary distribution of the personalized PageRank.

? Can you provide an explanation why implementing PageRank with (biased) restarts is more efficient in the suggested manner as opposed to performing calculations based on the explicit interpolated state transition matrix? (Hint: think of memory efficiency primarily.)

? Can you explain why does the probability in the stationary distribution for state 3 drop more then that of state 4 when we perform the personalized PageRank algorithm with restarts over states $\{1, 2\}$?

8.4 *Hubs and Authorities*

The **Hubs and Authorities**[11] algorithm (also referred as the **Hyperlink Induced Topic Search** or **HITS algorithm**) resembles PageRank in certain aspects, but there are also important differences between the two. The HITS and PageRank algorithms are similar in that both of them determine some importance scores to the vertices of a complex network. The HITS algorithm, however, differs from PageRank in that it calculates two scores for each vertex and that it operates directly on the adjacency matrix of the network.

[11] Kleinberg 1999

The fundamental difference between PageRank and HITS is that while the former assigns a single importance sore to each vertex, HITS characterizes every node from two perspectives. Putting the HITS algorithm into the context of web browsing, a website can become prestigious by

1. directly providing relevant contents or

2. providing valuable links to websites that offer relevant contents.

The two kind of prestige scores are tightly coupled with each other since a page is deemed as providing relevant contents if it is referenced by websites that are deemed as referencing relevant websites. Conversely, a website is said to reference relevant pages if the direct hyperlink connections it has point to websites with relevant contents. As such, the two kinds of relevance scores mutually depend on each other. We call the first kind of relevance as the **authority** of a node, whereas we call the second type of relevance as the **hubness** of a node.

The authority score of some node $j \in V$ is defined as

$$\boldsymbol{a}_j = \sum_{(i,j)\in E} \boldsymbol{h}_i. \tag{8.7}$$

Hubness scores are for node $j \in V$ are defined analogously as

$$\boldsymbol{h}_j = \sum_{(j,k)\in E} \boldsymbol{a}_k. \tag{8.8}$$

Recall, however, that the calculation of the authority scores for node $j \in V$ involves a summation over its incoming edges, whereas its hubness score depends on the authority scores of its neighbors accessible by outgoing edges.

The way we can calculate these two scores is pretty reminiscent to the calculation of the PageRank scores, i.e., we follow an iterative algorithm for that. Likewise to the calculation of PageRank scores, we also have a **globally convergent** algorithm for calculating the hubness and authority scores of the individual nodes of the network.

One difference compared to the PageRank algorithm is that HITS relies on the adjacency matrix during the calculation of the hub and authority scores of the nodes instead of a row stochastic transition matrix that is used by PageRank. The **adjacency matrix** is a simple square binary matrix $A \in \{0,1\}^{n\times n}$ which contains whether there exists a directed edge for all pairs of vertices in the network. An $a_{ij} = 1$ entry indicates that there is a directed edge between node i and j. Analogously, $a_{ij} = 0$ means that no directed edge exist between node i and j.

For the globally convergent solution of (8.7) and (8.8) we have $\boldsymbol{h}^* = \xi A\boldsymbol{a}^*$ and $\boldsymbol{a}^* = \nu A^\intercal \boldsymbol{h}^*$ for some appropriately chosen scalars ξ and ν. What it further implies is that for the solutions we have

$$\begin{aligned} \boldsymbol{h}^* &= \xi\nu AA^\intercal \boldsymbol{h}^* \\ \boldsymbol{a}^* &= \nu\xi A^\intercal A\boldsymbol{a}^*. \end{aligned} \tag{8.9}$$

According to (8.9), we can obtain both the ideal hubness and authority vectors as an eigenvector for the matrices $AA^\intercal$ and $A^\intercal A$, respectively. The problem with this kind of solution is that even though matrices – and their respective adjacency matrix – are typically sparse, by forming the **Gram matrices** $AA^\intercal$ or $A^\intercal A$, we would likely obtain matrices that are no longer sparse. The sparsity of matrices, however, is something that we typically value and we want to sacrifice it seldomly. In order to overcome this issue, we rather employ an asynchronous strategy for obtaining $\boldsymbol{h}^*$ and $\boldsymbol{a}^*$.

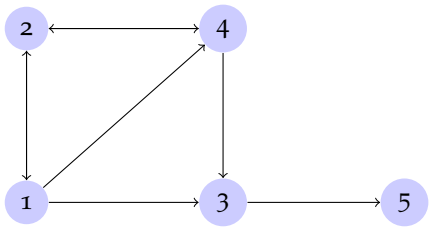
Algorithm 4 gives the pseudocode for HITS following the principles of asynchronicity. Line 5 and 7 of Algorithm 4 reveals that after each matrix multiplication performed in order to get an updated solution for $\boldsymbol{h}$ and $\boldsymbol{a}$, we employ a normalization step by ensuring that the largest element within these vectors equal to one. Without these normalization steps, the values in $\boldsymbol{h}$ and $\boldsymbol{a}$ would increase without bound and the algorithm would diverge. The normalization step divides each element of vectors $\boldsymbol{h}$ and $\boldsymbol{a}$ by the largest included in them.

Algorithm 4: Pseudocode for the HITS algorithm.

Require: adjacency matrix A
Ensure: vectors $\boldsymbol{a}$, $\boldsymbol{h}$ storing the authority and hubness of the vertices of the network

```
1:  function HITS(A)
2:      h = 1                          // Initialize the hubness vector to all ones
3:      while not converged do
4:          a = A^T h
5:          a = a/max(a)               // normalize a such that all its entries are ≤ 1.
6:          h = Aa
7:          h = h/max(h)
8:      end while
9:      return a, h
10: end function
```

Table 8.5 contains the convergence of HITS that we get when applying Algorithm 4 over the sample network structure presented in Figure 8.10. We can see that vertices indexed as 3 and 5 are totally useless as hubs. This is not surprising as a vertex with a good hubness score needs to link to vertices which have high authority scores.



(a) Example network.

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(b) Adjacency matrix of the network.

Figure 8.10: Example network to perform HITS algorithm on.

$\mathbf{h}^{(0)}$	$\mathbf{a}^{(1)}$	$\mathbf{h}^{(1)}$	$\mathbf{a}^{(2)}$	$\mathbf{h}^{(2)}$	$\mathbf{a}^{(3)}$	$\mathbf{h}^{(3)}$	...	$\mathbf{a}^{(10)}$	$\mathbf{h}^{(10)}$
1	0.5	1	0.3	1	0.24	1	...	0.21	1
1	1	0.5	1	0.41	1	0.38	...	1	0.36
1	1	0.17	1	0.03	1	0.007	...	1	0
1	1	0.67	0.9	0.69	0.84	0.71	...	0.79	0.72
1	0.5	0	0.1	0	0.02	0	...	$3,5e-07$	0

Table 8.5: Illustration of the convergence of the HITS algorithm for the example network included in Figure 8.10. Each row corresponds to a vertex from the network.

Even though vertex 3 has an outgoing edge, it links to such a vertex which is not authoritive. The case of vertex 5 is even simpler as it has no outgoing edges at all, hence it is completely incapable of linking to vertices with high authority scores (in particular it even fails at linking to vertices with *any* non-zero authority score).

Vertex 5 turns out to be a poor authority as well. This is in accordance to the fact that it receives a single incoming edge from vertex 3 which has been assigned a low hubness score.

Although vertex 3 is one of worst hubs, it is also one of the vertices with the highest authority score at the same time. Indeed, these are vertices 2 and 3 that obtain the highest authority scores. It is not surprising at all that vertices 2 and 3 have the same authority scores, since they have the same incoming edges from vertices 1 and 4. As a consequence of vertices 2 and 3 ending up to be highly authoritive, those vertices that link them manage to obtain high hubness. Since those vertices that are directly accessible from vertex 1 are a superset of those of vertex 4, the hubness score of vertex 1 surpasses that of vertex 4. As the above example suggests, the $\boldsymbol{a}$ and $\boldsymbol{h}$ scores from the HITS algorithm got into an equilibrial state even after a few number of iterations.

8.5 Further reading

PageRank and its vast number of variants, such as [12], admittedly belong to the most popular approaches of network analysis. As mentioned previously, we can conveniently model a series of real-world processes and phenomena with the help of complex networks, including but not limited to social interactions [13], natural language [14]

[12] Wu et al. 2013, Mihalcea and Tarau 2004, Jeh and Widom 2002

[13] Newman 2001, Hasan et al. 2006, Leskovec et al. 2010, Backstrom and Leskovec 2011, Matakos et al. 2017, Tasnádi and Berend 2015

[14] Erkan and Radev 2004, Mihalcea and Tarau 2004, Toutanova et al. 2004

or biomedicine [15].

[15] Wu et al. 2013, Ji et al. 2015

Liu [2006] provides a comprehensive overview of web data mining and Manning et al. [2008] offers a thorough description of information retrieval techniques.

8.6 Summary of the chapter

Many real world phenomena can be conveniently modeled as a network. As these networks have a potentially enormous number of vertices, efficient ways of handling these datasets is of utmost importance. Efficiency is required both in accordance of the storage of the networks and the algorithms employed over them.

In this chapter we reviewed typical ways, networks can be represented and multiple algorithms that assign relevance score to the individual vertices of a network. Readers are expected to understand the working mechanisms behind these algorithms.

9 | Clustering

Learning Objectives:

- The task of clustering
- Difference between agglomerative and partitioning approaches
- Hierarchical clustering techniques
- Improving hierarchical clustering
- k-means clustering
- Bradley-Fayyad-Reina algorithm

This chapter provides an introduction to clustering algorithms. Throughout the chapter, we overview two main paradigms of clustering techniques, i.e., algorithms that perform clustering in a hierarchical and partitioning manner. Readers will learn the about the task of clustering itself, understand their working mechanisms and develop an awareness of their potential limitations and how to mitigate those.

9.1 What is clustering?

Clustering deals with a partitioning of datasets into coherent subsets in an unsupervised manner. The lack of supervision means that these algorithms are given a dataset of observations *without* any target label that we would like to be able to recover or give accurate predictions for based on the further variables describing the observations. Instead, we are only given the data points as m-dimensional observations and we are interested in finding such a grouping of the data points such that those that are *similar to each other* in some – previously unspecified sense – would be grouped together. Since similarity of the data points is an important aspect of clustering techniques, those techniques discussed previously in Chapter 4 and Chapter 5 would be of great relevance for performing clustering.

As a concrete example, we could perform clustering over a collection of novels based on their textual contents in order to find the ones that has a high topical overlap. As another example, users with similar financial habits could be identified by looking at their credit card history. Those people identified as having similar behavior could then be offered the same products by some financial institution, or even better special offers could be provided for some dedicated cluster of people with high business value.

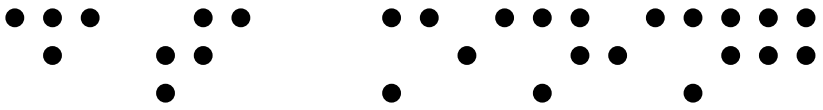
9.1.1 *Illustrating the difficulty of clustering*

As mentioned earlier, the main goal of clustering algorithms is to find groups of coherently behaving data points based on the commonalities in their representation. The main difficulty of clustering is that coherent and arguably sensible subsets of the input data can be formed in multiple ways. Figure 9.1 illustrates this problem. If we were given the raw dataset as illustrated in Figure 9.1 (a), we could argue that those data points that are located on the same y-coordinate form a cluster as depicted in Figure 9.1 (b). One could argue that a similar clustering which assigns points with the same x-coordinate into the same cluster would also make sense.

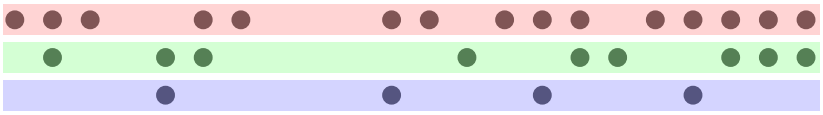
It turns out, however, that the '*true*' distribution of our data follows the one illustrated in Figure 9.1 (c) as the clusters within the data correspond to letters from the **Braille alphabet**. The difficulty of clustering is that the *true* distribution is never known in reality. Indeed, if we already knew the underlaying data distribution there would be no reason to perform clustering in the first place. Also, this example illustrates that the relation which makes the data points belong together can sometimes be a subtle non-linear one.

? Can you decode what is written in Figure 9.1 (c)?

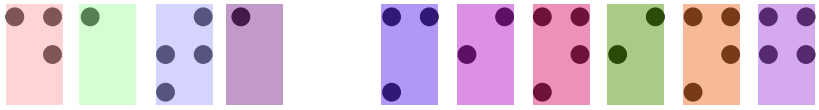
Clustering is hence not a well determined problem in the sense that multiple different solutions could be obtained for the same input data. Clustering on the other hand is of great practical importance as we can find hidden (and hopefully valid) structure within datasets.



(a) The raw unlabeled data.



(b) One sensible clustering of the data based on their y-coordinates.



(c) The real structure of the data.

Figure 9.1: A schematic illustration of clustering.

9.1.2 *What makes a good clustering?*

Kleinberg imposed three desired properties regarding the behavior of clustering algorithms[1]. The three criteria were **scale invariance**, **completeness** and **consistency**.

[1] Kleinberg 2002

What these concepts mean for a set of data points S and an associated distance over them $d : S \times S \to \mathbb{R}^+ \cup \{0\}$ are described below. Let us think of the output of some clustering algorithm as a function $f(d, S)$ which – when provided by a notion of pairwise distances over the points from S – provides a disjoint partitioning of the dataset S that we denote by Γ. Having introduced these notations, we can now revisit the three desiderata introduced by Kleinberg.

Scale invariance means that the clustering algorithm should be insensitive for rescaling the pairwise distances and provide the exact same output for the same dataset if the notion of pairwise distances change by a constant factor. That is, $\forall d, \alpha > 0 \Rightarrow f(d, S) = f(\alpha d, S)$. What it means intuitively, that imagining that our observations are described by vectors that indicate the size of a certain object, the output of the clustering algorithm should not differ if we provide our measurements in millimeters or if we provide them in miles or centimeters.

The **richness** property requires that if we have the freedom of changing d, i.e., the notion of pairwise distances over S, the clustering algorithm f should be able to output all possible partitioning of the dataset. To put it more formally, $\forall \Gamma \exists d : f(d, S) = \Gamma$.

The **consistency** criterion for a clustering algorithm f requires the output of f to be the same whenever the d is modified by a Γ-transformation. A **Γ-transformation** is such a transformation over some distance d and a clustering function f, such that the distances d' obtained by the Γ-transformation are such that the distances between pairs of points

- assigned to the same cluster by f do not increase,
- assigned to a different cluster by f do not decrease.

A clustering fulfils consistency, if for any d' obtained by a Γ-transformation we have $f(d, S) = \Gamma \Rightarrow f(d', S) = \Gamma$.

Although these properties might sound intuitive, Kleinberg pointed out that it is impossible to construct such a clustering algorithm f that would meet all the three criteria at the same time. Constructing clustering algorithms that meet two out of the previous desiderata is nonetheless feasible and this is the best we can hope for.

9.2 *Agglomerative clustering*

The first family of clustering techniques we introduce is **agglomerative clustering**. The way these clustering algorithms work is that they initially assign each and every data point into a cluster of its own then they gradually start merging them together until all the clusters belong into a single cluster. This bottom-up strategy builds up a hierarchy based on the arrangement of the data points and this is why this kind of approach is also referred to as **hierarchic clustering**.

The pseudocode for agglomerative clustering is provided in Algorithm 5. It illustrates that in the beginning every data point is assigned to a unique cluster which then get merged into a hierarchical structure by repeated mergers of pairs of clusters based on their inter-cluster distance. Applying different strategies for determining the inter-cluster distances could produce different clustering outcomes. Hence, an important question is how do we determine these inter-cluster distances that we shall discuss next.

Algorithm 5: Pseudocode for agglomerative clustering.

Require: Data points $\mathcal{D}$
Ensure: Hierarchic clustering of $\mathcal{D}$
1: **function** AGGLOMERATIVECLUSTERING($\mathcal{D}$)
2: $i = 0$
3: **for** $d \in \mathcal{D}$ **do**
4: $i = i + 1$
5: $C_i = \{d\}$ // each data point gets assigned to an individual cluster
6: **end for**
7: **for** (k=1; k < i; ++k) **do**
8: $[C_i^*, C_j^*] = \underset{(C_i, C_j) \in C \times C}{\arg\min} \; d(C_i, C_j)$ // find the closest pair of clusters
9: $C_k = C_i^* \cup C_j^*$ // merge the closest pair of clusters
10: **end for**
11: **end function**

9.2.1 *Strategies for merging clusters*

A key component for agglomerative clustering is how we select the pair of clusters to be merged in each step (cf. line 8 of Algorithm 5). Chapter 4 provided a variety of distances that can be used to determine the dissimilarity between a *pair of individual points*. We would, however, require a methodology which assigns a distance for a *pair of clusters*, with each clusters possibly consisting of *multiple data points*. The choice of this strategy is important as different choices for calculating inter-cluster distances might result in different result.

We could think of the inter-cluster distance as a measure which

tells us the cost of merging a pair of clusters. In each iteration of agglomerative clustering, we are interested in selecting the pair of clusters with the lowest cost of being merged. There are multiple strategies one can follow when determining the inter-cluster distances. We next review some of the frequently used strategies.

Let us assume that C_i and C_j denotes two clusters, each of which refers to a set of m-dimensional points. Additionally, we have a distance function d that we can use for quantifying the distance for any pair of d-dimensional data points.

Complete linkage performs a pessimistic calculation for the distance between a pair of clusters as it is calculated as

$$d(C_i, C_j) = \max_{\boldsymbol{x}_i \in C_i, \boldsymbol{x}_j \in C_j} d(\boldsymbol{x}_i, \boldsymbol{x}_j),$$

meaning that the distance it assigns to a pair of clusters equals to the distance between the pair of most distant points from the two clusters.

Single linkage behaves oppositely to complete linkage in that it measures the cost of merging two clusters as the smallest distance between a pair of points from the clusters, i.e.,

$$d(C_i, C_j) = \min_{\boldsymbol{x}_i \in C_i, \boldsymbol{x}_j \in C_j} d(\boldsymbol{x}_i, \boldsymbol{x}_j).$$

The way **average linkage** computes the distance between a pair of clusters is that it takes the pairwise between all pairs of data points that can be formed from the members of the two clusters and simply averages these pairwise distances out according to

$$d(C_i, C_j) = \frac{1}{|C_i||C_j|} \sum_{\boldsymbol{x}_i \in C_i} \sum_{\boldsymbol{x}_j \in C_j} d(\boldsymbol{x}_i, \boldsymbol{x}_j).$$

A further option could be to identify the cost between a pair of clusters as

$$d(C_i, C_j) = \max_{\boldsymbol{x} \in C_i \cup C_j} d(\boldsymbol{x}, \boldsymbol{\mu}_{ij}),$$

where $\boldsymbol{\mu}_{ij}$ denotes the mean of the data points that we would get if we merged all the members of cluster C_i and C_j together, i.e.,

$$\boldsymbol{\mu}_{ij} = \frac{1}{|C_i| + |C_j|} \sum_{\boldsymbol{x} \in C_i \cup C_j} \boldsymbol{x}.$$

Ward's method[2] quantifies the amount of increase in the variation that would be caused by merging a certain pair of clusters. That is,

[2] Ward 1963

$$d(C_i, C_j) = \sum_{\boldsymbol{x} \in C_i \cup C_j} \|\boldsymbol{x} - \boldsymbol{\mu}_{ij}\|_2^2 - \left(\sum_{\boldsymbol{x} \in C_i} \|\boldsymbol{x} - \boldsymbol{\mu}_i\|_2^2 + \sum_{\boldsymbol{x} \in C_j} \|\boldsymbol{x} - \boldsymbol{\mu}_j\|_2^2 \right),$$

where $\boldsymbol{\mu}_{ij}$ is the same as before , $\boldsymbol{\mu}_i = \frac{1}{|C_i|} \sum_{\boldsymbol{x} \in C_i} \boldsymbol{x}$ and $\boldsymbol{\mu}_j = \frac{1}{|C_j|} \sum_{\boldsymbol{x} \in C_j} \boldsymbol{x}$. The formula applied in Ward's method can be equivalently expressed in the more efficiently calculable form of

$$d(C_i, C_j) = \frac{|C_i||C_j|}{|C_i| + |C_j|} \|\boldsymbol{\mu}_i - \boldsymbol{\mu}_j\|_2^2.$$

Applying Ward's method has the advantage that it tends to produce more even-sized clusters.

Obviously, not only the strategy for determining the aggregated inter-cluster distances, but the choice for function d – which determines a distance over a pair of data points – also plays a decisive role in agglomerative clustering. In general, one could choose any distance measure for that, which could potentially affect the outcome of the clustering. For simplicity, we assume it throughout this chapter that the distance measure that we utilize is just the standard Euclidean distance.

9.2.2 *Hierarchical clustering via an example*

We now illustrate the mechanism of hierarchical clustering for the example 2-dimensional dataset included in Table 9.1. As mentioned earlier, we would determine the distance between a pair of data points by relying on their Euclidean (ℓ_2) distance.

data point	location
A	$(-3, 3)$
B	$(-2, 2)$
C	$(-5, 4)$
D	$(1, 2)$
E	$(2, 2)$

Table 9.1: Example 2-dimensional clustering dataset.

Table 9.2 includes all the pairwise distances between the pairs of clusters throughout the algorithm. Since distances are symmetric, we make use of the upper and lower triangular part of the inter-cluster distance matrices in Table 9.2 to denote the distances obtained by complete linkage and single linkage strategies, respectively.

? Why is it so that the distances in the upper and lower triangular of the inter-cluster distance matrix in Table 9.2 (a) are exactly the same?

We separately highlight the cost for the cheapest cluster mergers for both the complete linkage and the single linkage strategies in the upper and lower triangular parts of the inter-cluster distance matrices in Table 9.2. It is also worth noticing that many of the values in Table 9.2 do not change between two consecutive steps. This is something we could exploit for making the algorithm more effective.

The entire trajectory of hierarchical clustering can be visualized by a **dendrogram**, which acts as a tree-structured log visualizing

	A	B	C	D	E
A	—	$\sqrt{2}$	$\sqrt{5}$	$\sqrt{17}$	$\sqrt{26}$
B	$\sqrt{2}$	—	$\sqrt{13}$	$\sqrt{9}$	$\sqrt{16}$
C	$\sqrt{5}$	$\sqrt{13}$	—	$\sqrt{40}$	$\sqrt{53}$
D	$\sqrt{17}$	$\sqrt{9}$	$\sqrt{40}$	—	$\boxed{\sqrt{1}}$
E	$\sqrt{26}$	$\sqrt{16}$	$\sqrt{53}$	$\boxed{\sqrt{1}}$	—

(a) 1st step

	A	B	C	DE
A	—	$\boxed{\sqrt{2}}$	$\sqrt{5}$	$\sqrt{26}$
B	$\boxed{\sqrt{2}}$	—	$\sqrt{13}$	$\sqrt{16}$
C	$\sqrt{5}$	$\sqrt{13}$	—	$\sqrt{53}$
DE	$\sqrt{17}$	$\sqrt{9}$	$\sqrt{40}$	—

(b) 2nd step

	AB	C	DE
AB	—	$\boxed{\sqrt{13}}$	$\sqrt{26}$
C	$\boxed{\sqrt{5}}$	—	$\sqrt{53}$
DE	$\sqrt{9}$	$\sqrt{40}$	—

(c) 3rd step

	ABC	DE
ABC	—	$\boxed{\sqrt{53}}$
DE	$\boxed{\sqrt{9}}$	—

(d) 4th step

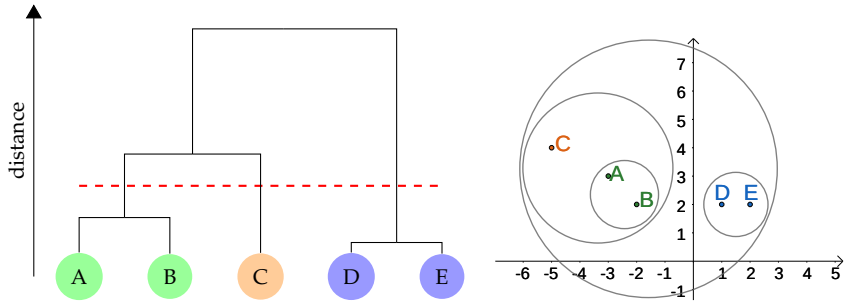
Table 9.2: Pairwise cluster distances during the execution of hierarchical clustering. The upper and lower triangular of the matrix includes the between cluster distances obtained when using complete linkage and single linkage, respectively. Boxed distances indicate the pair of clusters that get merged in a particular step of hierarchical clustering.

the cluster mergers performed during hierarchical clustering. Figure 9.2 (a) contains the dendrogram we get for the example dataset introduced in Table 9.1 when using Euclidean distance and the complete linkage strategy for merging clusters. The lengths of the edges in Figure 9.2 (a) are proportional to the inter-cluster distance that was calculated for the particular pair of clusters.

? How would the dendrogram differ if we performed different strategies for determining the inter-cluster distances, such as single linkage or average linkage?

Figure 9.2 (a) also illustrates a possible way to obtain an actual partitioning of the input data. We can introduce some fixed threshold – indicated by a red dashed line in Figure 9.2 (a) – and say that data points belonging to the same subtree after cutting the dendrogram at the given threshold would form a cluster of data points. The proposed threshold-based strategy would result in the cluster structure which is also illustrated in Figure 9.2 (b).

? Which of the nice properties introduced in Section 9.1.2 is not met by the threshold-driven inducing of clusters?



(a) The dendrogram structure. Distances are based on complete linkage.

(b) Geometric view of hierarchical clustering.

Figure 9.2: Illustration of the hierarchical cluster structure found for the data points from Table 9.1.

9.2.3 *Finding representative element for a cluster*

When we would like to determine a representative element for a collection of elements, we can easily take their **centroid** which simply corresponds to the averaged representations of the data points that belong to a particular cluster. There are cases, however, when averaging cannot be performed due to the peculiarities of the data. This could be the case when our objects are characterized by nominal attributes for instance.

The typical solution to handle this kind of situation is to determine the **medoid** (also called **clustroid**) of the cluster members instead of their centroids. The medoid is the element of some cluster C which lies the closest to all the other data points from the same cluster C in some *aggregated* sense (e.g. after calculating the sum or maximum of the within-cluster distances).

Example 9.1. *Suppose members of some cluster are described by the following strings:* $C = \{ecdab, abecb, aecdb, abcd\}$. *We would like to calculate the most representative element from that group, i.e., the member of the cluster that is the least dissimilar from the other members.*

When measuring the dissimilarity of strings, we could rely on the **edit distance** *(cf. Section 4.5). Table 9.3 (a) contains all the pairwise edit distances for the members of the cluster.*

Table 9.3 (b) contains the aggregated distances for each member of the cluster according to multiple strategies, i.e., summing, taking the maximum or the sum of squared distances of the within-cluster distances. According to any of the aggregations, it seems that the object aecdb *is the least dissimilar from the remaining data points in the given cluster, hence it should be treated as the representative element of that cluster.*

	ecdab	abecb	aecdb	abcd
ecdab	0	4	2	5
abecb	4	0	2	3
aecdb	2	2	0	3
abcd	5	3	3	0

(a) Pairwise distances between the cluster members.

	Sum	Max	Sum of squares
ecdab	11	5	45
abecb	9	4	29
aecdb	7	3	17
abcd	11	5	43

(b) Different aggregation of the within-cluster distances for each cluster member.

Table 9.3: Illustration of the calculation of the medoid of a cluster.

Example 9.1 might seem to suggest that the way we aggregate the within-cluster distances for obtaining the medoid of a cluster do not make a difference, i.e., the same object was the least dissimilar to all the other data points no matter whether we took the sum or the maximum or the squared sum of per-instance distances. Table 9.4 includes such an example within-cluster distance matrix for which

the way aggregation is performed makes a difference. Hence, we can conclude that aggregating the within-cluster distances differently, we could obtain different representative element for the same set of data points.

	A	B	C	D
A	0	3	1	5
B	3	0	4	3
C	1	4	0	6
D	5	3	6	0

(a) Pairwise within-cluster distances.

	Sum	Max	Sum of squares
A	**9**	5	35
B	10	**4**	**34**
C	11	6	53
D	14	6	70

(b) Different aggregation strategies.

Table 9.4: An example where matrix of within-cluster distances for which different ways of aggregation yields different medoids.

9.2.4 *On the effectiveness of agglomerative clustering*

The way agglomerative clustering works is that it initially introduces n distinct clusters, i.e., as many of them as many data points we have. Since we are merging two clusters in a time, we can perform $n-1$ merge steps before we find ourselves with a single gigantic cluster containing all the observations from our dataset.

In the first iteration, we have n clusters (one for each data points). Then in the second iteration, we have to deal with $n-1$ clusters. In general, during iteration i we have $n+1-i$ clusters to choose the most promising pair of clusters to merge. This is in line with the observation that in the last iteration of the algorithm (remember, we can perform $n-1$ merge steps at most) we would need to merge $n+1-(n-1)=2$ clusters together (which is kind of a trivial task to do).

Remember that deciding on the pair of clusters to be merged together can be performed in $O(k^2)$, if the number of clusters to choose from is k. Since

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6},$$

we get that the total computation performed during agglomerative clustering is

$$\sum_{i=1}^{n-1} (n+1-i)^2 = O(n^3).$$

Algorithms that are cubic in the input size are simply prohibitive for inputs that include more than a few thousands examples, hence they do not scale well to really massive datasets.

There is one way we could improve the performance of agglomerative clustering. Notice that the most of the pairwise distances calculated for the actual iteration can be reutilized during the next

iteration. To see why this is the case, recall that the first iteration requires the calculation of $O(n^2)$ pairwise distances and by the end of the first iteration we would end up having $n-1$ clusters as a results of merging a pair of clusters together.

We could then proceed by calculating all the pairwise distances for the $(n-1)$ clusters that we are left, but we can observe that if we did so, we would actually do quite much repeated work regarding the calculation of the distances for those pairs of clusters that we had already considered during the previous iteration. Actually, it would suffice to calculate a new distance of the single cluster that we just created in the last iteration towards all the others that were not involved in the last merging step. So, the agglomerative clustering would require the calculation of $n-2$ distances in its second iteration.

Storing inter-cluster distances in a **heap** data structure could hence improve the performance of agglomerative clustering in a meaningful way. The good property of heaps that they offer $O(\log h)$ operations for insertion and modification with h denoting the number of elements stored in the heap. Since this time we would store pairwise inter-cluster distances in a heap, $h = O(n^2)$, i.e., the number of elements in our heap is upper-bounded by the squared number of data points. This means that every operation would be $O(\log n^2) = O(2\log n) = O(\log n)$. Together with the fact that the number of per iteration operations needed during agglomerative clustering is $O(n)$ and that the number of iteration performed is $O(n)$, we get that the total algorithm can be implemented in $O(n^2 \log n)$.

While $O(n^2 \log n)$ is a noticable improvement over $O(n^3)$, it is still insufficient to scale for such cases when we have hundreds of thousands of data points. In cases when $n > 10^5$ one could either combine agglomerative clustering with some approximate technique, such as the ones discussed in Chapter 5, or resort to more efficient clustering techniques to be introduced in the followings.

9.3 *Partitioning clustering*

Partitioning clustering follows a fundamentally different philosophy compared to agglomerative clustering. The way partitioning clustering differs from agglomerative clustering is that it does not determine a full hierarchy of the cluster structure. What partitioning clustering algorithms do instead is that they divide the datasets into disjoint partitions (the clusters) without telling us anything about the relation of the distinct partitions.

9.3.1 K-means clustering

The **k-means** algorithm[3] is one of the most popular data mining algorithms[4] due to its simplicity and effectiveness. The algorithm can be also motivated from a theoretical aspect as it is related to Expectation-Maximization[5].

[3] Macqueen 1967

[4] Wu et al. 2007

[5] Dempster et al. 1977

Expectation-Maximization is a technique to derive maximum likelihood estimates of probabilistic models when certain random variables are not observable. Training the parameters of popular models, such as **Gaussian Mixture Model**s or **Hidden Markov Model**s can also be performed based on this technique.

One of the important differences between k-means and agglomerative clustering algorithms is that the value for k – denoting the number of clusters to be identified – has to be chosen in advance in the k-means algorithm. In the case of agglomerative clustering, one could just build the entire dendrogram and obtain the clusters in multiple different ways and number of clusters.

The optimization problem k-means clustering strives to solve is

$$\min_{\mu_1,\ldots,\mu_k} \sum_{i=1}^{k} \sum_{\boldsymbol{x}\in C_i} \|\boldsymbol{x} - \boldsymbol{\mu}_i\|_2^2. \tag{9.1}$$

That is, we would like to see the sum of squared differences between cluster centroids (denoted by $\boldsymbol{\mu}_i$) and data points assigned to the corresponding cluster C_i to be minimized. We perform a hard assignment of the data points $\boldsymbol{x} \in \mathcal{D}$ to the clusters based on the distance of $\boldsymbol{x}$ to the individual cluster centroids. To put it formally, $\boldsymbol{x} \in C_{i^*}$ such that

$$i^* = \min_i \|\boldsymbol{x} - \boldsymbol{\mu}_i\|_2. \tag{9.2}$$

To express (9.2) in simple terms, every data point is assigned to the cluster that is described by the closest centroid $\boldsymbol{\mu}_i$ to the data point $\boldsymbol{x}$. The way this simple rule can be actually interpreted is that we assume every cluster C_i to behave as a multivariate Gaussian distribution with a unique centroid $\boldsymbol{\mu}_i$ and a covariance matrix that is identical across all the clusters. For simplicity, we can assume that every cluster gets described by a multivariate Gaussian distribution having the identity matrix as their covariance matrix.

Under the previous assumptions,

$$\|\boldsymbol{x} - \boldsymbol{\mu}_i\|_2 < \|\boldsymbol{x} - \boldsymbol{\mu}_j\|_2$$

implies the inequality

$$p(\boldsymbol{x}|\boldsymbol{\mu}_i, \Sigma_i) > p(\boldsymbol{x}|\boldsymbol{\mu}_j, \Sigma_j),$$

i.e., the probability density function for data point $\boldsymbol{x}$ is higher with for cluster C_i than to C_j. Hence the distance $\|\boldsymbol{x} - \boldsymbol{\mu}_i\|_2$ provides a good proxy and a theoretically sound way to assign data points to clusters (assuming the clusters can be described by multivariate Gaussian distributions with the same covariance matrix).

It is worth mentioning that Gaussian Mixture Model (GMM) behave similarly to k-means, except for the fact that it utilizes a soft assignment of the data points to the different clusters. These soft assignments come in the form of distributions quantifying the extent to which a data point is believed to belong to the different clusters (defined as Gaussian distributions). GMM also differs from k-means algorithm in that it does not require the multivariate Gaussian distributions describing our clusters to be described by the same covariance matrix.

The pseudocode for k-means which follows the abovementioned considerations is illustrated in Algorithm 6. That is, we first (randomly) choose k initial cluster centroids, then we iteratively repeat the following steps:

- we assign all data points $\boldsymbol{x} \in \mathcal{D}$ to the cluster represented by the centroid closest to the given point (according to (9.2)),
- update the centroids for all clusters by taking the mean of the data points assigned to the given cluster.

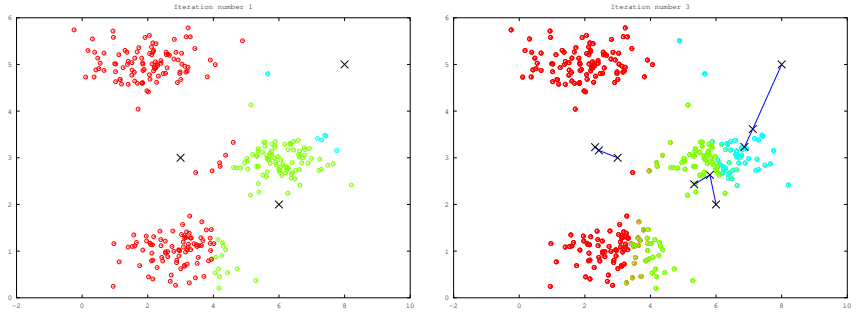
The results of performing Algorithm 6 with $k = 3$ on an example dataset is illustrated in Figure 9.3. We can see how do the initially randomly chosen cluster centroids get updated over the iterations of the algorithm. Data points in Figure 9.3 are colored based on the cluster they get assigned to over the different iterations. We can see it in Figure 9.3 that the change in the location of the cluster centroids decreases as the algorithm is making progress. Indeed, by the end of iteration 7, there clustering stabilizes. We should add it, however, that the solution k-means converges could vary on the choice of the initial location for the cluster centroids.

Figure 9.4 provides an illustration on the sensitivity of the k-means algorithm to the choice of the initial cluster centroids. Figure 9.4 displays the clusters we obtained with k-means (using $k = 10$) when applied over the Braille dataset from Figure 9.1 using different random initializations for the cluster centroids. We can see that the three runs resulted in three notably different outputs with different quality. Points within the grey boxes form a coherent group, i.e., a symbol from the Braille alphabet. Hence, we would ideally like to see data points within each grey area to be assigned to the same cluster, which would be illustrated graphically by those points being colored by the same color.

Require: Dataset $\mathcal{D} \in \mathbb{R}^{n \times m}$, k for the number of expected clusters
Ensure: $\mu \in \mathbb{R}^{k \times m}$ including the centroids for the k clusters

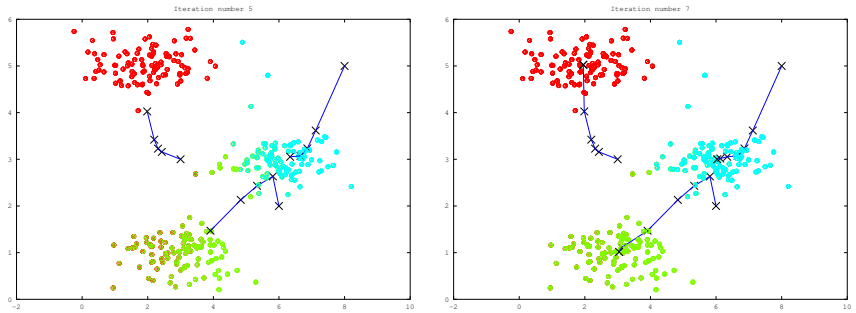
function KMEANSCLUSTERING($\mathcal{D}$, k)
  Initialize k centroids: $\mu = [\boldsymbol{\mu_1}, \ldots, \boldsymbol{\mu_k}]$
  while (stopping criterion is not met) **do**
    $\mu_{new} := zeros(k, m)$
    $clusterSizes := zeros(1, k)$
    for $i = 1$ to n **do**
      $c := \arg\min_j \|\boldsymbol{x}_i - \boldsymbol{\mu}_j\|^2$
      $\mu_{new}(c) := \mu_{new}(c) + \boldsymbol{x}_i$
      $clusterSizes(c) := clusterSizes(c) + 1$
    end for
    for $i = 1$ to k **do**
      $\mu_{new}(i) = \mu_{new}(i) / clusterSizes(i)$
    end for
    $\mu = \mu_{new}$
  end while
  return μ
end function

Algorithm 6: Pseudocode for the k-means algorithm.



(a) Cluster centroids and data points assignments after iteration 1

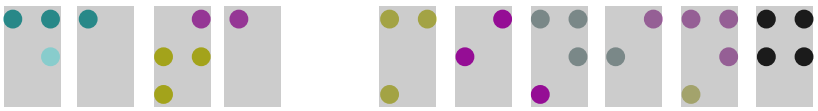
(b) Cluster centroids and data points assignments after iteration 3



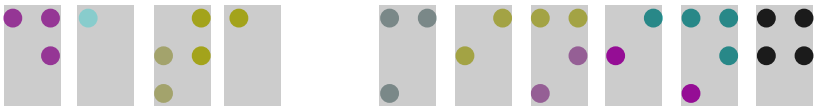
(c) Cluster centroids and data points assignments after iteration 5

(d) Cluster centroids and data points assignments after iteration 7

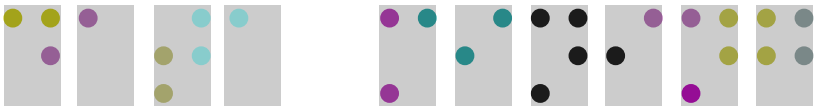
Figure 9.3: An illustration of the k-means algorithm on a sample dataset with $k = 3$ with the trajectory of the centroids being marked at the end of different iterations.



(a) Results of k-means with random initialization #1.



(b) Results of k-means with random initialization #2.



(c) Results of k-means with random initialization #3.

Figure 9.4: Using k-means clustering over the example Braille dataset from Figure 9.1. Ideal clusters are indicated by grey rectangles.

9.3.2 *Limitations of the k-means algorithm*

The simplicity and the computational efficiency makes the usage of the k-means algorithm an appealing choice. At the same time, it is important to know about its potential limitations. We have already mentioned a few of those, now we overview them and mention a few others as well.

One disadvantage of k-means is that it requires the number of clusters to be provided in advance. This naturally limits the richness of the clusters one can obtain by applying this kind of clustering technique.

Another limitation of the algorithm is that it might not necessarily converge to the global optima of its objective introduced in (9.1). What it means is that choosing the initial cluster centroids differently could lead us to find radically different clusters for the same dataset.

k-means algorithm implicitly assumes that the clusters behave as multivariate Gaussian distributions. Furthermore, there is an even stronger assumption about the covariance matrices of the different Gaussian distributions responsible for the description of the clusters being identical. As such, k-means could have a poor performance if these assumptions are violated. A partly related issue is that k-means algorithm is highly sensitive to outliers, i.e., atypical data points in the dataset.

A common heuristic that mitigates some of the earlier mentioned problems is to choose the initial cluster centroids from the actual data points such that the minimum distance between the selected cluster centroids gets maximized. The behavior of this heuristic is illustrated in Example 9.2.

Example 9.2. *If we perform k-means clustering over the example dataset introduced in Table 9.1 with $k = 3$, we would choose the three initial cluster centroids to be the data points B, C and E.*

First, we could choose data point C to be the first cluster centroid. This is an appealing choice as that point has the most extreme coordinate for both dimensions. Looking back at the pairwise distances provided in Table 9.2 (a), we could see that data point E is the furthest one from C (with a distance of $\sqrt{53}$), hence E would be selected as the second centroid. Now the third cluster center needs to be selected such that its closest distance to the already selected centroids C and E is the maximal.

For the three remaining candidates, we have that A, B and D has these values as $\min(\sqrt{5}, \sqrt{26})$, $\min(\sqrt{13}, \sqrt{16})$ and $\min(\sqrt{40}, \sqrt{1})$, respectively. Out of the three options B would behave the best for becoming the next cluster centroid which behaves the most dissimilar to the already chosen ones.

9.3.3 *Clustering streaming data — the Bradley-Fayyad-Reina algorithm*

A final difficulty could arise when we need to cluster streaming data. Datasets containing **streaming data** are more challenging to be processed because they have the special property that the full inventory of data points cannot be accessed at once. In the case of streaming data, we receive data points in an on-line fashion, pretty much we observe products passing by at some point of a conveyor belt. Data originating from sources such as Twitter or Facebook typically belong to the category of streaming data.

Algorithms operating over streaming data are called **streaming algorithms** that require special considerations due to the peculiarities of the data they operate on. First of all, they cannot assume all the data points to be accessible at the same time. Secondly, they have to be very efficient both in terms of speed and memory consumption, due to the possibly high throughput of the data stream.

The **Bradley-Fayyad-Reina (BFR) algorithm**[6] can be regarded as such an extension of k-means which was especially tailored for being applied on streaming datasets.

[6] Bradley et al. 1998

The way BFR algorithm works at the high level is that it keeps track of a number of clusters in the m-dimensional space and when a new data point arrives, it decides whether the data point is worth bein assigned to any of the already existing clusters. If the current data point is sufficiently dissimilar to all the existing clusters and does not fit well enough into any of them, we can create a new cluster for the currently received data point.

BFR makes the decision about which already created m-dimensional cluster does some data point $\boldsymbol{x}$ fits in the most by calculating the

quantity

$$\sum_{j=1}^{m} \frac{(\boldsymbol{x}_j - \boldsymbol{\mu}_{i,j})^2}{\boldsymbol{\sigma i, j^2}}, \tag{9.3}$$

where $\boldsymbol{\mu}_{i,j}$ and $\boldsymbol{\sigma i, j^2}$ denotes the j-th component of the coordinate-wise mean and variance for cluster i.

Notice that calculation in (9.3) for some cluster i is essentially equivalent to

$$(\boldsymbol{x} - \boldsymbol{\mu_i})^\intercal \Sigma_i^{-1} (\boldsymbol{x} - \boldsymbol{\mu_i}) \tag{9.4}$$

assuming that cluster i has mean vector $\boldsymbol{\mu_i}$ and a diagonal covariance matrix of the form $\Sigma_i = I\sigma_i^2$, where $I \in \mathbb{R}^{m \times m}$ denotes the identity matrix and $\boldsymbol{\sigma}_i^2 \in \mathbb{R}^m$ contains the coordinate-wise variances. We can also identify the expression in (9.4) as the **Mahalanobis distance** that we introduced in Section 4.2. Hence, what BFR does is to calculate the Mahalanobis distance between an incoming data point and the clusters that we describe by such special multivariate Gaussian distributions that have a diagonal covariance matrix.

Treating the covariance matrix of the Gaussian distributions that characterize our clusters to be diagonal means that we assume that there is no correlation between the different coordinates. This is obviously a compromise in modeling that we make for reducing the computational and memory footprint related to the calculation of cluster memberships. The advantage of this assumption is that it now suffices to keep track of m per coordinate mean and variance scores for each cluster. Additionally, the extent to which a data point belongs to a certain cluster can also be obtained by m subtractions and divisions. Had we allow the clusters to have an arbitrary covariance structure, we would need to store $m \times m$ covariance matrices and inverting those for each of the clusters.

The question which arises next is how can we *efficiently* keep track of the per coordinate variances of such samples, i.e., the clusters, which might eventually be expanded over time. Figure 9.5 provides a reminder on the sample **variance** for a random variable and how it can be expressed as a difference of expected values derived from the sample. Example 9.3 illustrates calculating the sample variance of a random variable according to the given methodology.

Example 9.3. *Let us take some sample of a random variable X as $[3, 5, 3, -1]$. Relying on the calculations included in Table 9.5 (a) we would obtain for the biased sample variance the result*

$$\frac{1}{4}\sum_{i=1}^{n}(x_i - \mu)^2 = \frac{0.25 + 6.25 + 0.25 + 12.25}{4} = 4.75.$$

MATH REVIEW | VARIANCE AS A DIFFERENCE OF EXPECTATIONS

The biased sample variance calculated over an n observations from a random variable X, i.e., from $x_1, x_2, \ldots, x_n$ is

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2,$$

with μ denoting the mean of the sample.
This expression, however, can be expressed in an alternative form as

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - 2\mu\frac{1}{n}\sum_{i=1}^{n} x_i + n\frac{1}{n}\mu^2.$$

Since $\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$, we have

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - \left(\frac{1}{n}\sum_{i=1}^{n} x_i\right)^2,$$

that is the difference between the expectation of the random variable X^2 and the squared expected value of X, where the random variable X^2 is derived from X by squaring the values of X.

Figure 9.5: Variance as a difference of expectations

When using the alternative calculation which is based on the difference of expectations, we could rely on the calculations included in Table 9.5 (b). In this case we would get that the biased sample variance is

$$\frac{1}{4}\sum_{i=1}^{n} x_i^2 - \left(\frac{1}{4}\sum_{i=1}^{n} x_i\right)^2 = 11 - 2.5^2 = 4.75.$$

sample	x_i	$x_i - \mu$	$(x_i - \mu)^2$
1	3	0.5	0.25
2	5	2.5	6.25
3	3	0.5	0.25
4	-1	-3.5	12.25
Average	2.5	0.0	4.75

(a) Calculations for determining variance according to its definition.

sample	x_i	x_i^2
1	3	9
2	5	25
3	3	9
4	-1	-1
Average	2.5	11

(b) Calculations for determining variance as a difference of expectations.

Table 9.5: Calculations involved in the determination of biased sample variance in two different ways for the observations $[3, 5, 3, -1]$.

Based on the methodology for efficiently calculating a running variance for such a sample of a random variable that could expand over time, let us consider the working mechanism of the BFR algorithm over a concrete example.

data point	location
A	$(-3, 3)$
B	$(-2, 2)$
C	$(-5, 4)$
D	$(1, 2)$
E	$(2, 4)$

Table 9.6: Example 2-dimensional clustering dataset for illustrating the working mechanism of the Bradley-Fayyad-Reina algorithm.

Example 9.4. *Suppose we have a dataset consisting of five points at coordinates as described in Table 9.6. We have formed two clusters so far, i.e.,* $C_1 = \{A, B, C\}$ *and* $C_2 = \{D, E\}$*. BFR maintains a separate* $2m + 1$*-dimensional representation for each of these clusters, where* $m = 2$ *is the dimensionality of our dataset.*

The first value in the summary vector for each cluster contain the number of data points that has been assigned to the given cluster so far. The remaining $2m$ *coordinates are for the coordinate-wise sum and the coordinate-wise sum of squares of the data points that are members of a particular cluster. Table 9.7 (a) summarizes the BFR representations for our clusters* C_1 *and* C_2*.*

The succinct representations of the clusters provided in Table 9.7 (a) let us know that the important statistics for the clusters are

- $\mu_1 = \left(\frac{-10}{3} \quad \frac{9}{3}\right)$ *and* $\sigma_1^2 = \left(\frac{38}{3} - \frac{100}{9} \quad \frac{29}{3} - \frac{81}{9}\right) = \left(1.55 \quad 0.67\right)$,
- $\mu_2 = \left(1.5 \quad 2\right)$ *and* $\sigma_2^2 \left(\frac{5}{2} - \frac{9}{4} \quad 10 - 3^2\right) = \left(0.25 \quad 1.0\right)$.

Based on these findings, we could easily figure out which cluster should a newly received data point, such as $F = (-1, 4)$ *be assigned to. All we have to do, is to plug in the coordinates of the data point and the values determined earlier into the formula* (9.4).

If we do so, we get that the value determined for C_1 *is lower than the score that we get towards* C_2*, i.e., we obtain scores* 5 *and* 26*. As such, we would decide to update cluster* C_1 *by data point* F*. Table 9.7 (b) contains the updated cluster representations after doing so.*

9.4 Further reading

There are additional approaches that are – nonetheless beyond the scope of this note – worth to be aware of. **Clustering Using REpresentatives**[7] (CURE) is a method mitigating some of the shortcomings of the k-means algorithm, while preserving its computational efficiency. One of the advantages of CURE is that it is an easily parallelizable algorithm.

[7] Guha et al. 1998

Spectral clustering[8] is based on the eigenvectors of similarity matrices derived from the data points. Spectral clustering can also

[8] Ng et al. 2001

Cluster id i	$\lvert C_i \rvert$	Sum of coordinates		Sum of squared coordinates	
1	3	-10	9	38	29
2	2	3	6	5	20

(a) The BFR representation of the clusters.

Cluster id i	$\lvert C_i \rvert$	Sum of coordinates		Sum of squared coordinates	
1	4	-11	13	39	45
2	2	3	6	5	20

(b) The BFR representation of the clusters after updating cluster C_1 with data point $(-1, 4)$.

Table 9.7: Illustration of the BFR representations of clusters $C_1 = \{A, B, C\}$ and $C_2 = \{D, E\}$ for the data points from Table 9.6.

be applied to perform partitioning of networks, i.e., determining coherent subset of vertices within graphs. One serious drawback of spectral clustering, however, compared to k-means is that it is much less efficient both in terms of memory consumption and speed.

Density-based spatial clustering of applications with noise (**DBSCAN**) [9] is another frequently applied clustering technique which is especially designed to be tolerant to the presence of outliers in the dataset.

[9] Ester et al. 1996

9.5 *Summary of the chapter*

This chapter introduced the problem of clustering where we are interested in identifying coherent subgroups of our input data points without any explicit signal for the kind of patterns we are looking for. We have introduced two important paradigms for performing clustering, i.e., agglomerative clustering and partitioning clustering.

Agglomerative clustering provides us a hierarchy of the cluster structure based on the inter-cluster (dis)similarities. Partitioning techniques, however, simply assign each data point into a cluster without providing a structural relation between the different clusters it identifies.

Standard clustering algorithms, such as k-means, is not designed to handle streaming data. At the end of the chapter, we introduced the Bradley-Fayyad-Reina (BFR) algorithm which can be regarded as an extension of the k-means algorithm with the capability of handling streaming data.

10 | Conclusion

We have covered some of the most fundamental algorithms of data mining. During our discussion, we repeatedly emphasized the fact that modern datasets are often so enormous that they could not necessarily be stored in the main memory of computers. To this end, a variety of techniques – including the application of approximate solutions – have been discussed to provide efficient algorithms that can cope with input datasets of large size as well.

A | Appendix — Course information

DESCRIPTION OF THE SUBJECT – MASTER LEVEL

(1.) Title: **Data mining**	**Credits: 3+2**
Category of the subject: compulsory	
The ratio of the theoretical and practical character of the subject: 60-40 (credit%)	
Type of the course: lecture + practice seminar **The total number of the contact hours**: 28 + 28 (lecture + practice seminar) during the semester **Language**: English	
Evaluation: • written exam at the end of the year for the lecture • mid-term tests for the practice where the students are allowed to use a computer	
The term of the course: I. semester	
Prerequisites (if any): --	

Description of the subject

General description:
The course focuses on prominent algorithms and topics of data mining with an emphasis on discussing the scalability of the algorithms. The course also introduces important concepts of data mining and puts them in broader (e.g. information theoretic) contexts. The main topics of the course are listed below:

- Introduction, attribute types and selection
- Various distance concepts and their efficient approximation techniques
- Dimension reduction algorithms
- Frequent pattern mining
- Graph-based data mining
- Web data mining, recommendation systems and text mining
- Agglomerative and partitioning clustering techniques
- Processing of data streams

Objectives and expected outcome

The aim of the course is to provide an advanced introduction to modern data mining techniques which pay a special attention of handling massive amounts of datasets. As such, students should develop the ability of designing and analysing fast (approximate) algorithms. During the course, students develop an in-depth understanding of various data mining concepts by gaining hands-on experience in data processing and implementing various algorithms.

Selected bibliography (2-5) (author, title, edition, ISBN)

Leskovec, Jure, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge university press, 3rd Edition (2020). ISBN: 9781108476348.

Liu, Bing. *Web data mining: exploring hyperlinks, contents, and usage data*. Springer Science & Business Media, 1st Edition (2007). ISBN: 978-3540378815

Tan, Pang-Ning, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*, 1st Edition (2006). ISBN: 978-0321321367

List of General competences (knowledge, skills, etc., *KKK **8.***) promoted by the subject

a) Knowledge

Students will

- be familiar with the most important concepts and definitions of data mining
- understand and be able to properly use the terminology of data mining
- develop and in-depth understanding of the most popular data mining algorithms and see their connections to other related fields.

b) Skills

Students will be able to

- design and employ data mining algorithms to real-world problems of massive sizes
- understand new scientific ideas from papers related to data mining
- participate in research projects with an understanding of the underlying mathematical concepts
- interpret the outputs of their work in a sound way.

c) Attitude

Students will be

- ready to understand the mathematical models involved in data mining algorithms
- aware and sensitive regarding the ethical aspects of using data mining algorithms
- open to cooperate with other researchers and working groups
- interested in new results, techniques and methods.

d) Autonomy and responsibility

Students will be

- able to schedule and organize their work independently
- aware towards the ethical usage of data mining techniques
- able to able to understand research ideas related to data mining independently.

Special competences promoted by the subject:

Knowledge	**Skills**	**Attitude**	**Autonomy/responsibility**
Students will learn basic concepts of data mining	Students will be able to implement data mining algorithms	Students will be open to study and apply new methods related to data mining	Students will develop self-determination for studying new concepts
Students will get to know information theory related concepts related to data mining	Students will be comfortable using numerical analysis software (Octave)	Students will be develop a critical way of thinking	Students will become aware of the ethical aspects of applying data mining algorithms
Students will learn about relevant concepts from probability theory	Students will be able to interpret the output of data mining algorithms	Students will develop interest in new models, algorithms and applications	Students will learn the importance of developing efficient data processing implementations
Students will know how to model real-world phenomena by mathematical structures such as graphs	Student will be able to choose the most appropriate data mining algorithms for some real-world modeling problem	Students will gain a data-oriented perspective for processing real-world problems	
Students will learn ways how to obtain valuable knowledge from massive amounts of data sets, including data streams	Students will be able to understand the working mechanisms of scalable data mining algorithms		

Instructor of the course (*name, position, scientific degree*): Dr. Gábor Berend, assistant professor, PhD
Teacher(s) (*name, position, scientific degree*): Dr. Gábor Berend, assistant professor, PhD

Bibliography

Galen Andrew, Raman Arora, Jeff Bilmes, and Karen Livescu. Deep canonical correlation analysis. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1247–1255, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL http://proceedings.mlr.press/v28/andrew13.html.

Lars Backstrom and Jure Leskovec. Supervised random walks: Predicting and recommending links in social networks. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM '11, pages 635–644, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0493-1. DOI: 10.1145/1935826.1935914. URL http://doi.acm.org/10.1145/1935826.1935914.

Ella Bingham and Heikki Mannila. Random projection in dimensionality reduction: Applications to image and text data. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pages 245–250, New York, NY, USA, 2001. ACM. ISBN 1-58113-391-X. DOI: 10.1145/502512.502546. URL http://doi.acm.org/10.1145/502512.502546.

Ingwer Borg, Patrick J.F. Groenen, and Patrick Mair. *Applied Multidimensional Scaling*. Springer Publishing Company, Incorporated, 2012. ISBN 3642318479, 9783642318474.

Magnus Borga. Canonical correlation a tutorial, 1999.

P. S. Bradley, Usama Fayyad, and Cory Reina. Scaling clustering algorithms to large databases. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, KDD'98, page 9–15. AAAI Press, 1998.

Euisun Choi and Chulhee Lee. Feature extraction based on the Bhattacharyya distance. *Pattern Recognition*, 36(8):1703 – 1709,

2003. ISSN 0031-3203. DOI: https://doi.org/10.1016/S0031-3203(03)00035-9. URL http://www.sciencedirect.com/science/article/pii/S0031320303000359.

A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, 39(1):1–38, 1977.

Michel Marie Deza and Elena Deza. *Encyclopedia of Distances*. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-00234-2_1.

John W. Eaton, David Bateman, Søren Hauberg, and Rik Wehbring. *GNU Octave version 4.2.0 manual: a high-level interactive language for numerical computations*, 2016. URL http://www.gnu.org/software/octave/doc/interpreter.

Günes Erkan and Dragomir R. Radev. LexRank: Graph-based lexical centrality as salience in text summarization. *J. Artif. Int. Res.*, 22(1):457–479, December 2004. ISSN 1076-9757. URL http://dl.acm.org/citation.cfm?id=1622487.1622501.

Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

Casper B. Freksen, Lior Kamma, and Kasper Green Larsen. Fully understanding the hashing trick. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 5389–5399. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/7784-fully-understanding-the-hashing-trick.pdf.

Kristina Gligoric, Ashton Anderson, and Robert West. How constraints affect content: The case of twitter's switch from 140 to 280 characters. In *Proceedings of the Twelfth International Conference on Web and Social Media, ICWSM 2018, Stanford, California, USA, June 25-28, 2018.*, pages 596–599, 2018. URL https://aaai.org/ocs/index.php/ICWSM/ICWSM18/paper/view/17895.

Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Cure: An efficient clustering algorithm for large databases. *SIGMOD Rec.*, 27(2):73–84, June 1998. ISSN 0163-5808. DOI: 10.1145/276305.276312. URL https://doi.org/10.1145/276305.276312.

Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, May 2000. ISSN 0163-5808. DOI: 10.1145/335191.335372. URL http://doi.acm.org/10.1145/335191.335372.

David R. Hardoon, Sandor R. Szedmak, and John R. Shawe-taylor. Canonical correlation analysis: An overview with application to learning methods. *Neural Comput.*, 16(12):2639–2664, December 2004. ISSN 0899-7667. DOI: 10.1162/0899766042321814. URL http://dx.doi.org/10.1162/0899766042321814.

Mohammad Al Hasan, Vineet Chaoji, Saeed Salem, and Mohammed Zaki. Link prediction using supervised learning. In *In Proc. of SDM 06 workshop on Link Analysis, Counterterrorism and Security*, 2006.

Harold Hotelling. Relations Between Two Sets of Variates. *Biometrika*, 28(3/4):321–377, 1936. ISSN 00063444. DOI: 10.2307/2333955. URL http://dx.doi.org/10.2307/2333955.

Glen Jeh and Jennifer Widom. SimRank: A measure of structural-context similarity. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 538–543, New York, NY, USA, 2002. ACM. ISBN 1-58113-567-X. DOI: 10.1145/775047.775126. URL http://doi.acm.org/10.1145/775047.775126.

Ming Ji, Qi He, Jiawei Han, and Scott Spangler. Mining strong relevance between heterogeneous entities from unstructured biomedical data. *Data Mining and Knowledge Discovery*, 29(4):976–998, Jul 2015. ISSN 1573-756X. DOI: 10.1007/s10618-014-0396-4. URL https://doi.org/10.1007/s10618-014-0396-4.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

Seyed Mehran Kazemi and David Poole. Simple embedding for link prediction in knowledge graphs. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 4289–4300. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/7682-simple-embedding-for-link-prediction-in-knowledge-graphs.pdf.

Jon Kleinberg. An impossibility theorem for clustering. In *Proceedings of the 15th International Conference on Neural Information Processing Systems*, NIPS'02, pages 463–470, Cambridge, MA, USA, 2002. MIT Press. URL http://dl.acm.org/citation.cfm?id=2968618.2968676.

Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, September 1999. ISSN 0004-5411. DOI: 10.1145/324133.324140. URL http://doi.acm.org/10.1145/324133.324140.

Dan Klien. Lagrange Multipliers Without Permanent Scarring. August 2004. URL www.cs.berkeley.edu/~klein/papers/lagrange-multipliers.pdf.

Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Rev.*, 51(3):455–500, August 2009. ISSN 0036-1445. DOI: 10.1137/07070111X. URL http://dx.doi.org/10.1137/07070111X.

J.B. Kruskal and M. Wish. *Multidimensional Scaling*. Sage Publications, 1978.

Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Predicting positive and negative links in online social networks. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 641–650, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8. DOI: 10.1145/1772690.1772756. URL http://doi.acm.org/10.1145/1772690.1772756.

Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, USA, 2nd edition, 2014. ISBN 1107077230.

Bing Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data (Data-Centric Systems and Applications)*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 3540378812.

J. Macqueen. Some methods for classification and analysis of multivariate observations. In *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715.

Antonis Matakos, Evimaria Terzi, and Panayiotis Tsaparas. Measuring and moderating opinion polarization in social networks. *Data Mining and Knowledge Discovery*, 31(5):1480–1505, Sep 2017. ISSN 1573-756X. DOI: 10.1007/s10618-017-0527-9. URL https://doi.org/10.1007/s10618-017-0527-9.

L. McInnes and J. Healy. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *ArXiv e-prints*, February 2018.

Rada Mihalcea and Paul Tarau. TextRank: Bringing order into texts. In Dekang Lin and Dekai Wu, editors, *Proceedings of EMNLP 2004*, pages 404–411, Barcelona, Spain, July 2004. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/W04-3252.

George A. Miller. Wordnet: A lexical database for english. *COMMUNICATIONS OF THE ACM*, 38:39–41, 1995.

Luis Carlos Molina, Lluís Belanche, and Àngela Nebot. Feature selection algorithms: A survey and experimental evaluation. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, pages 306–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1754-4. URL http://dl.acm.org/citation.cfm?id=844380.844722.

Fred Morstatter, Jürgen Pfeffer, Huan Liu, and Kathleen M. Carley. Is the sample good enough? comparing data from twitter's streaming api with twitter's firehose. In *Proceedings of the 7th International Conference on Weblogs and Social Media, ICWSM 2013*, pages 400–408. AAAI press, 2013.

M. E. J. Newman. The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences*, 98(2):404–409, 2001. DOI: 10.1073/pnas.98.2.404. URL http://www.pnas.org/content/98/2/404.abstract.

Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, page 849–856, Cambridge, MA, USA, 2001. MIT Press.

L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. In *Proceedings of the 7th International World Wide Web Conference*, pages 161–172, Brisbane, Australia, 1998. URL citeseer.nj.nec.com/page98pagerank.html.

Sam T. Roweis and Lawrence K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *SCIENCE*, 290:2323–2326, 2000.

Robert Speer, Joshua Chin, and Catherine Havasi. Conceptnet 5.5: An open multilingual graph of general knowledge. In *AAAI Conference on Artificial Intelligence*, 2016. URL http://arxiv.org/abs/1612.03975.

Jimeng Sun, Yinglian Xie, Hui Zhang, and Christos Faloutsos. Less is more: Sparse graph mining with compact matrix decomposition. *Statistical Analysis and Data Mining*, 1(1):6–22, 2008. DOI: 10.1002/sam.102. URL https://doi.org/10.1002/sam.102.

Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005. ISBN 0321321367.

Ervin Tasnádi and Gábor Berend. Supervised prediction of social network links using implicit sources of information. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, pages 1117–1122, Republic and Canton of Geneva, Switzerland, 2015. International World Wide Web Conferences Steering Committee. ISBN 978-1-4503-3473-0. DOI: 10.1145/2740908.2743037. URL http://dx.doi.org/10.1145/2740908.2743037.

Hanghang Tong, Spiros Papadimitriou, Jimeng Sun, Philip S. Yu, and Christos Faloutsos. Colibri: Fast mining of large static and dynamic graphs. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 686–694, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-193-4. DOI: 10.1145/1401890.1401973. URL http://doi.acm.org/10.1145/1401890.1401973.

Kristina Toutanova, Christopher D. Manning, and Andrew Y. Ng. Learning random walk models for inducing word dependency distributions. In Carla E. Brodley, editor, *ICML*, volume 69 of *ACM International Conference Proceeding Series*, New York, NY, USA, 2004. ACM.

Théo Trouillon, Christopher R. Dance, Éric Gaussier, Johannes Welbl, Sebastian Riedel, and Guillaume Bouchard. Knowledge graph completion via complex tensor factorization. *J. Mach. Learn. Res.*, 18 (1):4735–4772, January 2017. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=3122009.3208011.

Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008. URL http://www.jmlr.org/papers/v9/vandermaaten08a.html.

Jr. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58:236–244, 1963.

Martin Wattenberg, Fernanda Viégas, and Ian Johnson. How to use t-sne effectively. *Distill*, 2016. DOI: 10.23915/distill.00002. URL http://distill.pub/2016/misread-tsne.

Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1113–1120, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-516-1. DOI: 10.1145/1553374.1553516. URL http://doi.acm.org/10.1145/1553374.1553516.

Gang Wu, Wei Xu, Ying Zhang, and Yimin Wei. A preconditioned conjugate gradient algorithm for GeneRank with application to microarray data mining. *Data Mining and Knowledge Discovery*, 26 (1):27–56, Jan 2013. ISSN 1573-756X. DOI: 10.1007/s10618-011-0245-7. URL https://doi.org/10.1007/s10618-011-0245-7.

Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowl. Inf. Syst.*, 14(1):1–37, December 2007. ISSN 0219-1377. DOI: 10.1007/s10115-007-0114-2. URL http://dx.doi.org/10.1007/s10115-007-0114-2.

INDEX

Γ-transformation, 193
`ans` (variable), 22
`plot`, 26
`size` (function), 21

adjacency list, 170
adjacency matrix, 170, 187
agglomerative clustering, 194
anonymous function, 21
anti-monotonity, 140
aperiodicity, 180
association rule, 141
ATLAS, 24
authority, 187
average linkage, 195

Bhattacharyya coefficient, 68
Bhattacharyya distance, 68
BLAS, 24
bloom filter, 92
Bonferroni's principle, 36
Bradley-Fayyad-Reina algorithm, 205
Braille alphabet, 192
broadcasting, 22

candidate set generation, 146
centroid, 198
CERN, 32
characteristic matrix, 77
characteristic polynomial, 51
Chebyshev distance, 58
Cholesky decomposition, 45
city block distance, 58
classification, 47
closed frequent item set, 145
closed item set, 145
clustroid, 198
collaborative filtering, 121
complete linkage, 195
completeness, 193
complex network, 168
concept, 191
ConceptNet, 169
conditional FP-tree, 164
conditional probability, 35
confidence, 141
consistency, 193
constrained optimization, 108
contingency table, 46
corpus, 177
cosine distance, 62
cosine similarity, 62
covariance matrix, 43, 44, 111
CURE clustering, 208
curse of dimensionality, 104

d-sphere, 102
damping factor, 183
data mining, 31
DBSCAN clustering, 209
dead end, 181
dendrogram, 196
diagonalizable matrix, 119
Dice similarity, 65
distance metric, 55

dot operator, 21
dot product, 88
dummy variable trap, 39

edit distance, 66, 198
eigendecomposition, 119
eigenvalues, 50
eigenvectors, 50
elementwise calculation, 21
ergodicity, 180
Euclidean distance, 58
Expectation-Maximization, 201

false negative error, 85, 94
false positive error, 85, 94
feature discretization, 48
feature selection, 48
FP-Growth, 160
FP-tree, 160
frequent item set border, 145
frequent pattern mining, 138
frequent-pattern tree, 160
Frobenius norm, 120

gamma function, 102
Gaussian Mixture Model, 201
generalized eigenvalue problem, 133
geometric distribution, 183
geometric probability, 89
Gibbs inequality, 72
global convergence, 175, 187
Gram matrices, 188
Gramian matrix, 111

Hadamard product, 29
half-space, 89
hash set, 92
hashing trick, 66
heap, 200
Hellinger distance, 70
Hidden Markov Model, 201
hierarchic clustering, 194
HITS algorithm, 186
hubness, 187
Hubs and Authorities, 186
Hyperlink Induced Topic Search, 186
hypersphere, 101

identity matrix, 61
indexing, 177
information retrieval, 177
inner product, 88
Intel MKL, 24
irreducibility, 180

Jaccard similarity, 64
Jensen-Shannon divergence, 72
Johnson-Lindenstrauss lemma, 117

k-means, 201
Karush-Kuhn-Tucker conditions, 108
Kullback-Leibler divergence, 71

Lagrange function, 108
Lagrange multipliers, 108
lambda expressions, 21
LAPACK, 24
Large Hadron Collider, 32
law of total probability, 34
LDA, 131
lift, 142
Linear discriminant analysis, 131
log sum inequality, 72
longest common subsequence, 66

Mahalanobis distance, 60, 206
Manhattan distance, 58
Maple, 17
marginal probability, 35
MATLAB, 16, 17
matrix rank, 123

maximal frequent item set, 144
mean centering data, 40
medoid, 198
minhash signature, 80
minhash value, 77
Minkowski distance, 57
multicollinearity, 39
mutual information, 46

n-gram, 65
null space, 53
numerical analysis, 17
numpy, 29

Octave, 16, 17

PageRank, 177
Park–Chen–Yu algorithm, 155
partitioning clustering, 200
PCA, 105
Personalized PageRank, 185
pigeon hole principle, 93, 155
positive border, 145
positive semidefinite matrix, 45
power method, 173
principal component analysis, 105
principal eigenvalue, 173
principal eigenvector, 111
probabilistic data structure, 92
Python, 29

random projection, 117
random walk, 175
recommendation systems, 121
richness, 193
row stochastic matrix, 171

scale invariance, 193
scatter matrix, 44, 110, 111
scatter plot, 27
Scilab, 17
scipy, 29
semantic network, 169
Shannon entropy, 47
shingles, 65
single linkage, 195
Singular Value Decomposition, 118
sketch, 89
spectral clustering, 208
spider trap, 182
standardizing, 41
stationary distribution, 173
steady state distribution, 173
stochastic vector, 172
streaming algorithms, 205
streaming data, 205
support, 139
surjective function, 93
SVD, 118
symmetric matrix, 45

transposition, 67
truncated SVD, 124
TrustRank, 185

variance, 206
vectorization, 24

Ward's method, 195
whitening, 43, 60
WordNet, 169

z–score, 41