Gossip-Based Machine Learning in Fully Distributed Environments

István Hegedűs

Supervisor
Dr. Márk Jelasity

MTA-SZTE Research Group on Artificial Intelligence
and the University of Szeged

PhD School in Computer Science
University of Szeged

A thesis submitted for the degree of
Doctor of Philosophy

Szeged
2016

List of Algorithms

List of Figures

List of Tables

Contents

CHAPTER 1

Introduction

Data mining and machine learning algorithms are present in our digital life even if we do not recognize this. For example, these algorithms are the spam filtering methods in our e-mail client, the automatic moderation in a social site or in a blog, the autocompletion mechanisms in a web search engine or in a text editor and the recommendations on a movie or on a web-shop site. Other algorithms help us or protect our health in a medical application or just amuse us in a game. They can recognize spoken words and handwritten letters. Without a detailed description of its workings, let us quote a definition from Tom M. Mitchell for machine learning [83]:

> ″A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E“

On the basis of this definition, the goal is to learn from the examples. These examples are usually generated by us when we drag and drop a mail into the spam folder, when we click on a search result or an advertisement on a website or in a mobile application. These examples can form a database that can be used to train the machine learning algorithms to extract information from the data or identify patterns in the databases. This information and these patterns can help us to understand how things work around us, identify and predict the trends and optimize the traffic in a city.

Although the huge amount of data is usually stored on servers, clusters or clouds owned by firms, it was generated on our devices (PCs, laptops, tablets, smart phones, wearable devices, sensors in a smart house, etc.). The processing of this data is an even more challenging task due to its size and restricted accessibility. Parallel computing techniques can still handle the size problem, but as time goes by, the data accumulates and the companies need more and more resources to store and process it. They have to have bigger and bigger servers and data farms, which have sufficient space to store our data, sufficient computational capacity to serve our queries and sufficient memory to run data mining tasks. Moreover, the access to the collected data is often forbidden even to researchers. The motivation for using fully distributed (i.e. peer-to-peer (P2P)) machine learning algorithms partly comes from the above-mentioned reasons. Since the data is generated on the device that we use in our everyday life, these devices should work collectively and solve the data processing tasks together. Machine learning algorithms should be also trained on our devices as well in a distributed manner. The other motivation for the P2P algorithms is the privacy issue. Nobody wants their search history, private photos from a cloud or the list of movies that you saw ever to be leaked out just because a hacker has found a backdoor in a server. Distributed methods allow us to keep our private data in our device instead of uploading our files into data centers, but they can still build a machine learning model based on the data. Here, we demonstrate a possible method for fully distributed machine learning.

This thesis is organized as follows. First, we give an overview in Chapter 2, which summarizes the necessary background, includes an introduction to supervised learning, an outline of the applied system model and the data distribution; and we introduce the fully distributed algorithms through some examples. Later, in chapters 3 – 6, we present the main parts of this thesis, where we introduce a fully distributed learning scheme that can be applied with any data modeling algorithm that can be trained in an online manner. We present several learning algorithms that can provide these kinds of online models, including sophisticated models. Afterwards we present state-of-the-art machine learning algorithms such as boosting and matrix factorization. Then, we describe a modification of the framework for achieving higher efficiency and capability for concept drift handling.

The gossip learning approach that we propose in Chapter 3 involves models that perform random walks in the P2P network, and which are updated each time they visit a node, using the locally stored data. Here, there are as many models in the network as the number of nodes. Any online algorithm can be applied as a learning algorithm that is capable of updating models using a continuous stream of examples. Besides this, the proposed framework builds models on distributed data and it supports privacy preservation as well.

We present a boosting algorithm in Chapter 4, which also demonstrates the viability of gossip learning for implementing state-of-the-art machine learning algorithms. A boosting algorithm constructs a classifier incrementally by adding simple classifiers to a pool. The weighted vote of these classifiers determines the final classification. Here, we propose a pure online version of the FILTERBOOST boosting algorithm and we highlight the importance of model diversity in the P2P network.

In Chapter 5, we focus on the problem of concept drift, which occurs when the data patterns change. We propose two approaches to follow concept drift in the above mentioned framework. The first approach manages the distribution of the lifetime of the models in the network, hence we have both young (and thus adaptive) models and old models (which have a high performance) at all times. As our second contribution, we propose a mechanism that can detect the occurrence of concept drift by estimating and monitoring changes in the performance of the models.

In Chapter 6, we propose the singular value decomposition (SVD), an approach for the low-rank decomposition of a matrix A into low rank matrices (X, Y), which consist of orthogonal vectors. The low-rank and singular value decomposition of a matrix are important tools in data mining and they are widely used in areas such as recommender systems, graph clustering, dimension reduction and the topic modeling of documents. Here we present a stochastic gradient descent (SGD) algorithm to find the SVD, where matrices A and X are stored in

Table 1.1. The relationship between the chapters and the corresponding publications (where • and ◦ refer to the basic and the related publications, respectively).

	Chapter 3	Chapter 4	Chapter 5	Chapter 6
CCPE 2013 [89]	•	◦	◦	◦
EUROPAR 2012 [52]	◦	•		
SASO 2012 [53]	◦		•	
SISY 2012 [50]	◦		•	
ACS 2013 [51]	◦		•	
P2P 2014 [49]	◦			•
EUROPAR 2011 [88]	◦			
ICML 2013 [106]	◦			
ESANN 2014 [18]	◦			
TIST 2016 [47]	◦			◦
PDP 2016 [48]	◦			
PDP 2016 [8]	◦			◦

a fully distributed way and instances of the matrix Y perform random walks in the network. When a Y visits a node, it gets updated based on the local row of A, and the local row of X gets updated as well.

Finally, in Chapter 7 we summarize our contributions. Above, Table 1.1 depicts the relation between the relevant publications and the chapters [1].

[1]The implementation of the proposed algorithms in this thesis is available online at: https://github.com/isthegedus/Gossip-Learning-Framework

CHAPTER 2

Background

In this chapter we give a brief introduction to the problem we tackle, namely the problem of supervised machine learning. We describe an optimization mechanism for solving the above-mentioned problem. Then we describe our fully distributed system model and data distribution. After, we present an overview of some well-known fully distributed algorithms.

2.1 Supervised Learning

Throughout this thesis we address the problem of *supervised learning* in a fully distributed manner. For this, here we introduce the basic notations and the definition of supervised learning as follows. We are given a labeled data set that consists of the data examples or instances. Each instance is a feature vector with a corresponding class label generated by an unknown underlying probability distribution $\mathcal{D}$.

The above means that we are given objects which are described by features (or attributes). These objects belong to different classes. For example, consider the so-called "play tennis" database that was originally introduced in [94]. Here, we have to group days into positive and negative classes based on their weather properties (like temperature and wind speed). Another example might be the e-mail spam filtering problem, where we would like to decide whether the received mail is a spam mail or not. To describe an e-mail with a feature vector we can use the well-known bag-of-words method, which transforms the text into a binary vector.

More formally, let us denote this data set by $S = \{(x_1, y_1), \ldots, (x_n, y_n)\} \subset \mathbb{R}^d \times C$ where d is the *dimension* of the problem (the number of the features that describe an instance of the data set) and C is the domain of the class labels. In the case of *binary* classification the number of classes (K) is exactly 2, $C = \{0, 1\}$ (or using a different notation but having the same meaning $C = \{-1, 1\}$). In *multi-class* classification problems, where $K > 2$ (e.g. in the case of optical character recognition (OCR) the number of classes is the same as the number of possible letters to be recognized), $C = \{0, 1, \ldots, K\}$ (or $C = \{-1, 1\}^K$). That is, the class label may be a number that equals i if the data instance corresponds to the ith class; or a vector, where just the ith element has a value of 1 and the others have a value of -1. In summary, every element of the data set is represented by a feature vector (x) and the corresponding class label (y).

The main goal when solving a classification problem is to find the patterns or rules that explain why an object belongs to a specific class (why an e-mail become spam or not). In other words, the task is to identify how the representation of the object (the features) correlates with the class labels. A more precise description of the classification is that we seek a parametric function $f_w : \mathbb{R}^d \to C$ using the observations from S that can classify *any samples* including those that are not in the data set, but are also generated by the same probability distribution $\mathcal{D}$. This property is known as *generalization* and we would use this function to predict the label of yet unseen objects (decide whether the received mail is a spam, or what letters are on a scanned page). The function f_w is called the *model* of the data, and the vector w is the parameter of the model. These parameters are learned in the model training phase, or the parameters of the model are selected from a hypothesis space by a searching method (called the learning algorithm). When the data samples are available as a stream, the training process is known as *online learning*.

The labeled data set mentioned above is usually split into two disjoint sets called the *training* and *test* sets. The training set is applied in the model training process and the test set is used for model evaluation purposes only. Using this technique we can objectively compare the different learning algorithms with each other based on their performance on the test set.

2.2 Gradient Based Search

After introducing the problem of classification, we now describe an optimization method for solving the above-mentioned task (finding the parameters of a model).

The *gradient descent (GD)* method is an optimization method for seeking the local maximum (minimum) of a parametric function $F(z)$ by iteratively taking steps in the direction of the (negative) gradient. In other words, this method looks for the parameter z of function $F(z)$ where it has its local maxima (minima).

Let consider the problem of classification, where we look for the parameter vector w of a machine learning model f_w that allows the model to classify the instances of a data set most precisely. For this, we define the empirical risk (or objective) function

$$E(f_w) = \frac{1}{n}\sum_{i=1}^{n} l(\hat{y}_i, y_i), \tag{2.1}$$

where the $\hat{y} = f_w(x)$ is the lable *predicted* by the model of the sample (x, y) and l is a loss (or cost) function. The loss function measures the cost if the model predicts $\hat{y}$ for the *expected* label y of the sample x. To find the parameters of the model we have to find the minimum of the risk function ($E(f_w)$). The GD method can find these parameters by iteratively updating the parameters based on the gradient of the this function, which is computed using the training data set.

Here, different loss functions can be used (like squared loss) without loss of generality. The update rule for the parameter vector w at iteration t is

$$w_{t+1} = w_t - \eta\nabla_w E(f_w) = w_t - \eta\frac{1}{n}\sum_{i=1}^{n} \nabla_w l(f_w(x_i), y_i), \tag{2.2}$$

where η is the *learning rate* (the size of the gradient step) and n is the number of the training instances. With an appropriately chosen learning rate, the method will converge to a local optima of the objective function [21]. If this function is convex, then all local optima are also global optima, where the function has its minimal value.

The *stochastic gradient descent (SGD)* method uses a simplification, namely the update step is based on (and the gradient is computed on) a single (usually) uniform randomly selected training instance instead of computing the gradient on the whole training set:

$$w_{t+1} = w_t - \eta_t \nabla_w l(f_w(x_t), y_t) \tag{2.3}$$

For the convergence of the stochastic gradient method, the learning rate (η) has to satisfy the conditions: $\sum_t \eta_t^2 < \infty$ and $\sum_t \eta_t = \infty$ [21, 95]. If we assume that the training instances in a data stream follow a uniform random sampling from the underlying distribution, then the stochastic gradient method is a possible solution for the online classification problems as well.

In classification problems we look for the model that has the minimal error, as mentioned above, but the method may find a parameter vector that is very specific to the training data set, since the risk function is optimized on this set. This means that the model, after the training phase becomes more accurate on the training set, but it may have a worse performance measured on unseen samples and loses its generalization capability. When this happens it is called *overfitting*. A commonly applied technique to have a better generalization of the model is to add extra constraints to the parameter vector w in the objective function which are often called *regularization*. This extension can help the model to avoid overfitting on the training data set [20].

2.3 System Model and Data Distribution

As our system model and in general as a P2P system we consider a network of nodes (called *peers*) – where the number of nodes can be potentially very large – which are typically individual personal computing devices such as personal computers, laptops, smart phones, tablets and wearable devices. Each node in the network has a unique network address; and every node can communicate to other nodes through messaging if the address of the target node is locally available.

The messages in the network can be delayed or lost; moreover, nodes can leave and join the network again (*node churn*) without prior warning. The only assumptions are that the message delay has an upper bound and that the message drop rate is smaller than one. Of course these assumptions are also necessary conditions since the messages with an unbounded delay or with a drop probability that is one will never reach the receiver. Thus communication will halt among the nodes. Furthermore, we assume that when re-joining the network, a node has the same state as at the time of going offline.

Regarding data distribution, we assume that the records of the database are distributed horizontally; that is, all the nodes store full records (i.e. a d dimensional feature vector and the corresponding class label as well). In addition, we assume here that all the nodes store *exactly one record* (although the algorithms can be trivially adapted to a more general case and in fact profit from it, if a node has several data records).

Another important assumption is that the data never leaves the nodes; that is, collecting the data at a central location is not allowed. This assumption is motivated by *privacy preservation* concerns. This is especially important in smart phone applications [2, 70, 93], where the key motivation is to give the user full control over personal data. Also, collecting this data into a central server is prevented in systems like sensor and mobile ad hoc networks due to the physical constraints on communication.

In the applications, it is also common for a user to provide only a single record, like a personal profile, private documents, a search history, ratings, sensor values or location information of a smart phone. Having access to a single local record excludes the possibility of any local statistical processing of the whole data set, i.e. finding the maximal value of records or averaging them. Besides this, complex problems (like recommendation and spam filtering) require more advanced aggregations and models on the data.

Algorithm 2.1 Peer Sampling Service

```
1: view ← init()
2: loop
3:     wait($\Delta$)
4:     $p$ ← selectPeer()
5:     send view ∪ me to $p$
6: end loop

7: procedure ONRECEIVE(descriptors)
8:     view ← merge(view, descriptors)
9: end procedure
```

2.4 An Overview of Fully Distributed Algorithms

When using P2P algorithms as described above, given a network of nodes, these nodes usually run the same algorithm at any time and only with message passing solve global tasks together without any central control. Here, we have a communication graph that defines which nodes can send messages to each other and we call them *neighbors*. This graph is called the *overlay network* and is created by a *peer sampling service* that is also a distributed algorithm. Some protocols describe how we can build a specific overlay on the nodes, while others use the overlays as a service. Here, we describe algorithms that build overlays and algorithms that compute global functions in a fully distributed manner.

2.4.1 Peer Sampling

Since gossip-based algorithms and protocols work through message passing among the nodes in the network, we assume that an additional middleware (peer sampling service) is also available. These services can provide addresses of peers from the network. The addresses are stored locally and they were selected appropriately from the network. There are numerous peer sampling protocols [59, 60, 107] available and here we give an outline of two algorithms that highlight the usefulness of distributed algorithms.

These two methods are the T-MAN and the NEWSCAST protocols. Their algorithms are very similar, but they have a very different behavior and they result in different overlay networks. The skeleton of a general peer sampling service can be seen in Algorithm 2.1. Both the T-MAN and the NEWSCAST protocols manage a list of neighbors; these are node addresses and the corresponding descriptors called the *view*. Every peer in the network has a local view with size k. Initially, this view is filled by the addresses and descriptors of randomly selected nodes or by a bootstrap protocol. Every peer in the network performs the following steps periodically (the Δ at line 3 is the delay between two periods). It selects a peer taken from its local view (line 4), sends its local view and its own descriptor to the selected peer (line 5). If a node receives a list of descriptors, then it merges with its local view (line 8).

T-MAN In a part of the fully distributed algorithms, a peer should communicate with in some way similar peers from the network. In the P2P literature, a possible solution for building a similarity overlay is the T-MAN protocol that was published in [59]. This protocol extends the peer descriptors in Algorithm 2.1 by a similarity value which is computed between the descriptor of a neighbor and the descriptor of the current node. Next, after the merge the peer keeps the top k most similar neighbors. This way, this protocol converges to a similarity overlay.

NEWSCAST There are protocols that can provide addresses of probably available (online) peers at any time which are selected uniformly at random from the network. The NEWSCAST protocol is a peer sampling service that is capable of providing addresses of nodes that satisfy the conditions nemtioned previously. A fully distributed implementation of the protocol was published in [60, 107]. Here, the extension of the descriptor is a timestamp t. A node sends its own descriptor to the selected peer and sets t to the current time. After the merge of the lists, the node keeps the top k youngest neighbors. This protocol provides an overlay where the neighbors of nodes are uniformly selected and probably online.

2.4.2 Calculating Global Functions

The main goal of this thesis is to apply machine learning algorithms in a fully distributed environment. That is, to compute a global function on fully distributed data or data sets. Before we present our contributions in the next sections, we propose solutions for some basic problems.

Searching the minimal value The first example is the minimum search in a set of numbers. Let given n real numbers, chosen uniformly at random. Every

Algorithm 2.2 Searching Minimal Value

```
1: value ← init()
2: loop
3:     wait(Δ)
4:     p ← selectPeer()
5:     send value to p
6: end loop

7: procedure ONRECEIVE(rValue)
8:     value ← min(value, rValue)
9: end procedure
```

peer in the network has one of these numbers. The centralized solution is to ask the peers to send their number to the cloud, and in the cloud we can iterate on the numbers and find the minimum. But this problem can be solved in a fully distributed manner as well. The idea is the following. Every peer thinks that its number is the minimum in the network, called the value in Algorithm 2.2. A peer iteratively chooses one of its neighbors and sends its value to the selected neighbors. When a peer receives a number it checks that whether the received number is smaller than its own value. If it is, the peer changes the value to the received number and so on. After several sending phases every peer will have the minimum value of the network.

Other examples and solutions for more complex problems, such as searching maximal value, computing average or the PageRank, can be found in [57, 58, 63].

In the next chapters we present machine learning algorithms that are applicable in the above-described systems and are capable of solving complex problems.

CHAPTER 3

Gossip-Based Machine Learning

Here we first present gossip learning, a conceptually simple and powerful generic framework for designing efficient, fully distributed, asynchronous, local algorithms for learning models on fully distributed data. This framework is based on multiple models taking random walks over the network in parallel, while applying an online learning algorithm to improve themselves. Afterwards, we present a set of different machine learning algorithms that can be applied in this framework and we highlight the key parts of their implementation.

3.1 Related Work

In the area of P2P computing, a large number of fully distributed algorithms are known for calculating global functions over fully distributed data, generally referred to as aggregation algorithms. The literature of this field is vast, we mention only two examples, namely Astrolabe [109] and gossip-based averaging [58]. These algorithms are simple and robust, but are capable of calculating only simple functions such as the average. Nevertheless, these simple functions can serve as key components for more sophisticated methods, such as the EM algorithm [69], unsupervised learners [101] and the collaborative filtering-based recommender algorithms [14, 46, 87, 108]. However, here we seek to provide a rather generic approach that covers a wide range of machine learning models, while maintaining robustness and simplicity.

In the past few years there have been an increasing number of proposals for P2P machine learning algorithms as well, like those in [5, 6, 7, 28, 54, 77, 101]. The usual assumption in these studies is that a peer has a subset of the training data on which a model can be learned locally. After learning the local models, algorithms either aggregate the models to allow each peer to perform local predictions, or they assume that prediction is performed in a distributed way. Clearly, distributed prediction is a lot more expensive than local prediction; however, model aggregation is not needed, and there is more flexibility in the case of varying data.

3.2 Gossip Learning

The skeleton of the *Gossip Learning Framework (*GoLF*)* we propose is shown in Algorithm 3.3. This algorithm is run by every node in the network. When joining the network, the node generates an initial model as its CURRENTMODEL (in line 1). After the initialization each node starts to periodically send its current model to a neighbor, the time length of a period is Δ. The neighbor is selected using a peer sampling service (in line 4). As we mentioned in Section 2.3, we use the NEWSCAST gossip-based peer sampling protocol, which can provide node addresses selected uniformly at random from the network. When receiving a model, the node updates it (in line 8) based on the training sample (x, y) that it has locally, and subsequently stores the model (in line 9). The node selection mechanism results in the models taking random walks in the network and the update step improves the model in the meantime.

In GoLF, every model that is performing a random walk is theoretically guaranteed to converge so long as we assume that peer sampling service is working correctly. Since the nodes in the network store locally the received models (as its CURRENTMODEL or the latest models can be collected in a bounded queue), they can use them to predict the labels of new instances without additional commu-

Algorithm 3.3 Gossip Learning Framework

```
1: currentModel ← initModel()
2: loop
3:     wait($\Delta$)
4:     $p$ ← selectPeer()
5:     send currentModel to $p$
6: end loop

7: procedure onReceiveModel($m$)
8:     $m$.updateModel($x, y$)
9:     currentModel ← $m$
10: end procedure
```

nication cost. Moreover, incoming models can be combined as well, both locally (e.g., merging the received models, or implementing a local voting mechanism on the models in the queue) [88, 89] or globally (e.g., finding the best model in the network) [52].

We make no assumptions about either the synchrony of the periodic loops at the different nodes or the reliability of the messages. It is only assumed that the length of the period of the loop (called cycle) is the same (Δ) at all nodes. This framework can be applied for every machine learning model that can be trained in an online manner (instance by instance). The methods to be implemented are the INITMODEL() for creating a new initial learning model, the UPDATEMODEL() that performs the online update based on the training instance stored by the node and the PREDICT() function for label prediction.

Finally, a few words about the computation and communication costs of the framework. First, for the communication cost; each node in the network sends exactly one message in each Δ time unit (one cycle). The size of a message depends on the selected hypothesis space, which normally contains the parameters of a single model. For example a linear model in a binary classification scenario, which uses a linear discriminant function, needs $d + 1$ parameters that represent the separating hyperplane. In addition, the message also contains a small constant number of network addresses as defined by the NEWSCAST protocol (which is typically around 10 – 20). The computational cost of a model update depends on the selected online learner and the number of received messages by a node that follows a Poisson distribution with parameter $\lambda = 1$.

Algorithm 3.4 Logistic Regression

```
 1: procedure UPDATEMODEL($x, y$)
 2:     $\eta \leftarrow 1/(\lambda \cdot t)$
 3:     $err \leftarrow y - \text{prob}(x)$
 4:     $w \leftarrow (1 - \eta \cdot \lambda)w - \eta \cdot err \cdot x$
 5:     $t \leftarrow t + 1$
 6: end procedure

 7: procedure PROB($x$)
 8:     return $1/(1 + exp(w^T x))$
 9: end procedure

10: procedure INITMODEL
11:     $m.t \leftarrow 0$
12:     $m.w \leftarrow (0, \ldots, 0)^T$
13:     return $m$
14: end procedure

15: procedure PREDICT($x$)
16:     return round(prob($x$))
17: end procedure
```

3.3 Learning Components

In this section we describe several online learning algorithms that can be used in the above-mentioned framework. Furthermore, we give a possible implementation of the required methods for the different learning algorithms.

3.3.1 Logistic Regression

First, we present a commonly used classification algorithm, called the logistic regression method [20], which looks for the parameter vector w that maximizes the logarithm of the conditional data likelihood. The optimization function is given in Equation 3.1, where we applied a conventional technique to prevent w from having large values and which helps the model to achieve a better generalization. That is the so-called regularization.

$$\max_w \frac{1}{n} \sum_{i=1}^{n} y_i \log f_w(x_i) + (1 - y_i) \log(1 - f_w(x_i)) \qquad - \frac{\lambda}{2} ||w||^2 \tag{3.1}$$

To optimize the parameter vector w, we used the stochastic gradient method, which updates w with a uniform randomly chosen data point. In Algorithm 3.4 we presented the main functions that should be used for training a logistic regression model. Here, the UPDATEMODEL updates the model m based on the training sample (x, y); PROB returns the probability of x belonging to the positive ($y = 1$) class; finally, PREDICT returns the label of x based on the state of the

Algorithm 3.5 Pegasos SVM

```
1: procedure UPDATEMODEL($x, y$)
2:     $\eta \leftarrow 1/(\lambda \cdot t)$
3:     $w \leftarrow (1 - \eta \cdot \lambda)w$
4:     if $y \cdot w^T x < 1$ then
5:         $w \leftarrow w + \eta \cdot y \cdot x$
6:     end if
7:     $t \leftarrow t + 1$
8: end procedure
9: procedure INITMODEL
10:     $m.t \leftarrow 0$
11:     $m.w \leftarrow (0, \dots, 0)^T$
12:     return $m$
13: end procedure
14: procedure PREDICT($x$)
15:     return sign($w^T x$)
16: end procedure
```

current model.

3.3.2 Pegasos SVM

Next, we present an instantiation of an algorithm taken from the Support Vector Machines (SVM) [27] family. The SVM solvers are mainly applied in binary classification problems and look for a hyperplane which maximizes the *margin* that separates the examples of the positive and the negative classes.

$$\begin{aligned} \min_{w,b,\xi_i} \quad & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \xi_i \\ \text{s.t.} \quad & y_i(w^T x_i + b) \geq 1 - \xi_i \quad \text{and} \quad \xi_i \geq 0 \quad (\forall i:\ 1 \leq i \leq n) \end{aligned} \tag{3.2}$$

A formal description of this problem can be seen in Equation 3.2, where the parameter $w \in \mathbb{R}^d$ represents the normal vector of the separating hyperplane with the bias $b \in \mathbb{R}$. Here, ξ_i is the slack variable of the ith sample, which can be interpreted as the amount of misclassification errors of the ith sample, and C is a trade-off parameter between generalization and error minimization. The Pegasos algorithm [100] is an SVM training method that optimizes the above-described problem using the stochastic gradient descent approach. An implementation of the required methods for incorporating this algorithm into our framework can be seen in Algorithm 3.5, here $y \in \{-1, 1\}$.

Algorithm 3.6 Perceptron

```
1: procedure UPDATEMODEL($x, y$)
2:     $\eta \leftarrow 1/(\lambda \cdot t)$
3:     $grad \leftarrow (f(w^T x) - y) \cdot f'(w^T x)$
4:     $w \leftarrow (1 - \eta \cdot \lambda)w - \eta \cdot grad \cdot x$
5:     $t \leftarrow t + 1$
6: end procedure
7: procedure INITMODEL
8:     $m.t \leftarrow 0$
9:     $m.w \leftarrow (0, \dots, 0)^T$
10:    return $m$
11: end procedure
12: procedure PREDICT($x$)
13:    return $f(w^T x)$
14: end procedure
```

3.3.3 Perceptron

The perceptron learning algorithm is a binary classification method that optimizes the parameter w for separating two classes from each other. The mechanism was introduced by Rosenblatt [98].

Here, we present a general algorithm that optimizes the square of the error and uses regularization as well (see Equation 3.3). The function f is the so-called *activation function* (e.g. sigmoid, step, tanh) that is applied on the inner product of the parameter vector w and the instance vector.

$$\min_w \frac{1}{2n} \sum_{i=1}^{n} (f(w^T x_i) - y_i)^2 + \frac{1}{2}||w||^2 \tag{3.3}$$

The pseudo code of the corresponding procedures for the stochastic gradient training of the model is presented in the Algorithm 3.6.

3.3.4 Naive Bayes

The naive Bayes [83] machine learning algorithm is a probability-based classifier. It predicts the label for an instance that has the maximal conditional probability; namely,

$$\max_{y \in C} P(y|x) \tag{3.4}$$

Approximating this probability is quite difficult in practice. Because of this, the naive Bayes algorithm makes the assumption that the features are conditionally independent of each other. Applying this assumption, we have to approximate

Algorithm 3.7 Naive Bayes

```
 1: procedure UPDATEMODEL($x, y$)
 2:     $t \leftarrow t + 1$
 3:     $p$[y] $\leftarrow$ $p$[y]+1
 4:     $\mu$[y]$\leftarrow$ $\mu$[y]+$x$
 5:     $\sigma$[y]$\leftarrow$ $\sigma$[y]+$x^2$
 6: end procedure
 7: procedure PREDICT($x$)
 8:     $\mu \leftarrow \mu[i]/t; \sigma \leftarrow \sqrt{\sigma[i]/t - \mu^2}$
 9:     return $\text{argmax}_i p[i]/t \prod_{j=1}^{d} prob_{\mathcal{N}(\mu,\sigma)}(x_j)$
10: end procedure
11: procedure INITMODEL(K)
12:     $m.t \leftarrow 0$
13:     for i=0; i<K; i++ do
14:         $m.p$[i] $\leftarrow 0$
15:         $m.\mu$[i] $\leftarrow (0, \dots, 0)^T$
16:         $m.\sigma$[i] $\leftarrow (0, \dots, 0)^T$
17:     end for
18:     return $m$
19: end procedure
```

the probability values in Equation 3.5 based on our observations (the training instances) to solve the classification problem.

$$\max_{y \in C} P(y) \prod_{i=1}^{d} P(x_i|y) \tag{3.5}$$

In our algorithm (Algorithm 3.7) we made a further assumption, namely that the features follow normal distributions. Therefore we have to model these distributions by approximating their parameters (the mean – μ and the variance – σ) based on the training instances for all dimensions.

Naturally, for higher numerical precision the sum of the logarithmic probability values can be used instead of the product in Equation 3.5 and in the prediction function of the algorithm as well.

3.3.5 One vs. All Metaclassifier

In Algorithm 3.8 we present an extension of our framework that allows the use of binary classifiers for multi-class problems. This method solves the task that has K classes as K binary classification problems in the following way. An instance that has the label $0 \leq y < K$ will be a positive sample for the yth classifier and a negative sample for all other classifiers [20]. In the case of prediction, the label that will be assigned to the instance is the index of the classifier that classifies it the most reliably as a positive sample.

Algorithm 3.8 One vs. All Classifier

```
1: procedure UPDATEMODEL($x, y$)
2:     for i=0; i<K; i++ do
3:         if i==$y$ then
4:             $m$[i].update($x$,1)
5:         else
6:             $m$[i].update($x$,0)
7:         end if
8:     end for
9: end procedure
10: procedure INITMODEL(K)
11:     Models[] $m$
12:     for i=0; i<K; i++ do
13:         $m$[i]←initModel()
14:     end for
15:     return $m$
16: end procedure
17: procedure PREDICT($x$)
18:     return argmax$_i$ $m$[i].predict($x$)
19: end procedure
```

3.3.6 Artificial Neural Network

In our framework we also implemented another multi-class supervised learner method, namely the Artificial Neural Network (ANN). This algorithm can be interpreted as a directed acyclic network of perceptrons that are grouped into so-called layers. In general, an ANN contains an input layer, which has d (the dimension of the instances) neurons, some hidden layers and an output layer which has K (the number of classes) neurons. Increasing the number (using more hidden layers) or the size (using more neurons in the hidden layers) of the hidden layers allows the network to search in a higher hypothesis space, so it can increase the representation power of the network.

$$\max_{\Theta} \frac{1}{n} \sum_{i=1}^{n} \sum_{k=1}^{K} y_{i,k} \log f_{\Theta}(x_i)_k + (1 - y_{i,k}) \log(1 - f_{\Theta}(x_i)_k) \tag{3.6}$$

To train the neural network we used the SGD method as well, where the network propagates forward the training instance and computes the error at the output layer (e.g. the misclassification error). Afterwards, it updates the weights of the neurons by backpropagating this error to optimize the function in Equation 3.6 [20]. The corresponding functions of the learning algorithm can be seen in Algorithm 3.9. Here, the Θs represent the weights between the layers that should be optimized and the function f is the activation function. The weights are the part of the model and are initialized by taking uniform random numbers from $[0, 1]$. Here, we do not mention regularization, but it can also be used to

Algorithm 3.9 Artificial Neural Network

```
1: procedure UPDATE($x, y$)
2:     $\hat{y} \leftarrow$evaluate($x$)
3:     $\delta \leftarrow \hat{y} - y$
4:     for i=numLayers-1; i>0; i- - do
5:         grad$\leftarrow f(z^{i-1})^T\delta$
6:         $\delta \leftarrow \Theta^{(i)T}\delta \circ f'(z^{(i-1)})$
7:         $\Theta^{(i)} \leftarrow \Theta^{(i)} - \eta\cdot$grad
8:     end for
9:     grad$\leftarrow x\delta^T$
10:    $\Theta^{(0)} \leftarrow \Theta^{(0)} - \eta\cdot$grad
11: end procedure
12: procedure EVALUATE($x$)
13:     $z^{(0)} \leftarrow x^T\Theta^{(0)}$
14:     $a \leftarrow f(z^{(0)})$
15:     for i=1; i<numLayers; i++ do
16:         $z^{(i)} \leftarrow a^T\Theta^{(i)}$
17:         $a \leftarrow f(z^{(i)})$
18:     end for
19:     return $a$
20: end procedure
21: procedure PREDICT($x$)
22:     return argmax$_i$ evaluate($x$)$_i$
23: end procedure
```

handle the problem of overfitting.

3.4 Evaluating Algorithms

The most conventional method for evaluating machine learning algorithms is by spliting our database into two non-overlapping sets, called the training set and the test set. The training dataset is used to build the machine learning model while the test set is used to measure its performance. In the case of classification problems, the most common method for measuring the efficiency of a learning method is by counting the correct and the incorrect instance classifications on the test data set. To measure this, various methods are available in the literature, and some of them will be described below.

In our framework we implemented a mechanism for evaluating the models stored by the peers in the network. In the initialization phase we distribute only the training data set among the nodes in the network in the above-defined manner (every node gets exactly one instance). In every Δ time (cycle), every node in the network uploads the latest (last received and updated) models to a central machine to evaluate them. In this central machine the expected class labels (y) of the test data set is known and the PREDICT method of the model returns the predicted label ($\hat{y}$). Based on these labels we can use numerous evaluation metrics that can characterize the performance of the models.

In our framework we implemented the following evaluation metrics:

0-1 Error

$$E = \frac{1}{n}\sum_{i=1}^{n}\delta(y_i = \hat{y}_i),$$

where δ is the Kronecker delta.

Root Mean Square Error (RMSE)

$$E = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

These methods measure the degree of misclassification of a trained model on the test data set in two different ways.

Confusion Matrix

$$\begin{array}{c|c} \hat{y}\backslash y & \begin{matrix} 0 & 1 \end{matrix} \\ \begin{matrix} 0 \\ 1 \end{matrix} & \begin{pmatrix} TN & FN \\ FP & TP \end{pmatrix} \end{array}$$

Here TN means *true negative*; that is, the number of test instances that were correctly classified as a negative sample. FN is the *false negative*, FP is the *false positive* and TP is the *true positive* metrics, respectively. Furthermore y means the expected label of an instance and $\hat{y}$ is predicted by using the trained model. Obviously, this evaluation method can easily be extended to more than two classes and other interesting metrics can be derived as well [36, 103].

3.5 Experiments

Here, we present the effectiveness of the proposed framework by applying the Pegasos SVM as the learning algorithm. In our experiments, we used the PEERSIM event-based P2P simulator [84].

3.5.1 Experimental Setup

Data Sets. We used three different data sets; namely Reuters [45], Spambase, and the Malicious URLs [78] data sets, which were obtained from the UCI database

Table 3.1. The main properties of the data sets and the prediction error (0-1 error) of the baseline sequential algorithm. In the case of Malicious URLs dataset the results of the full feature set are shown in parentheses.

	Reuters	SpamBase	Malicious URLs (10)
Training set size	2,000	4,140	2,155,622
Test set size	600	461	240,508
Number of features	9,947	57	10
Class label ratio	1,300:1,300	1,813:2,788	792,145:1,603,985
Pegasos 20,000 iter.	0.025	0.111	0.080 (0.081)

repository [12]. These data sets are of different types including small and large sets containing a small or large number of features. Table 3.1 shows the main properties of these data sets, as well as the prediction performance of the Pegasos algorithm.

The original Malicious URLs data set has a huge number of features ($\sim$ 3,000,000), therefore we first performed a feature reduction step so that we can carry out simulations. Note that the message size in our algorithm depends on the number of features, so in a real application this step might also be useful in such extreme cases. We applied the well-known correlation coefficient method for each feature with the class label, and kept the ten features with the maximal absolute values. If necessary, this calculation can also be carried out in a gossip-based fashion [58], but we performed it offline. The effect of this dramatic reduction on the prediction performance is shown in Table 3.1, where the results of the Pegasos algorithm on the full feature set are shown in parenthesis.

Evaluation metric. The evaluation metric we focus on is prediction error. To measure prediction error, we need to split the datasets into training sets and test sets. The proportions of this splitting are shown in Table 3.1. In our experiments, we track the misclassification ratio over the test set of 100 randomly selected peers. The misclassification ratio of a model is simply the number of the misclassified test examples divided by the number of all test examples, which is the so called 0-1 error.

Modeling failure. In a set of experiments we model extreme message drop and message delay. Drop probability is set to be 0.5. This can be considered an extremely large drop rate. Message delay is modeled as a uniform random delay from the interval [Δ, 10Δ], where Δ is the gossip period in Algorithm 3.3. This is also an extreme delay, orders of magnitudes higher than what can be expected in a realistic scenario, except if Δ is very small. We also model realistic churn based on probabilistic models in [105]. Accordingly, we approximate online session length with a lognormal distribution, and we approximate the parameters of the distribution using a maximum likelihood estimate based on a trace from a private BitTorrent community called FileList.org obtained from Delft University of Technology [97]. We set the offline session lengths so that at any moment in time 90% of the peers are online. In addition, we assume that when a peer comes back online, it retains its state that it had at the time of leaving the network.

3.5.2 Results

The experimental results for the Pegasos algorithm are shown in Figure 3.1. Note that all variants can be mathematically proven to converge to the same result, so the difference is in convergence speed only. As can be seen, the algorithm variants converge to the performance of the sequential method (horizontal dashed line). Moreover, since the nodes can remember the models that pass through them at no communication cost, we can cheaply implement a simple voting mechanism, where nodes will use more than one model to make predictions. This voting mechanism improves the speed of the convergence of the algorithm, in our experiments we used a cache of size 10.

Figure 3.1 also contains results from our extreme failure scenario. We can observe that the difference in convergence speed is mostly accounted for by the increased message delay. The effect of the delay is that all messages wait 5 cycles on average before being delivered, so the convergence is proportionally slower. In addition, half of the messages get lost too, which adds another factor of about 2 to the convergence speed. Apart from slowing down, the algorithms still converge to the correct value despite the extremely unreliable environment, as was expected.

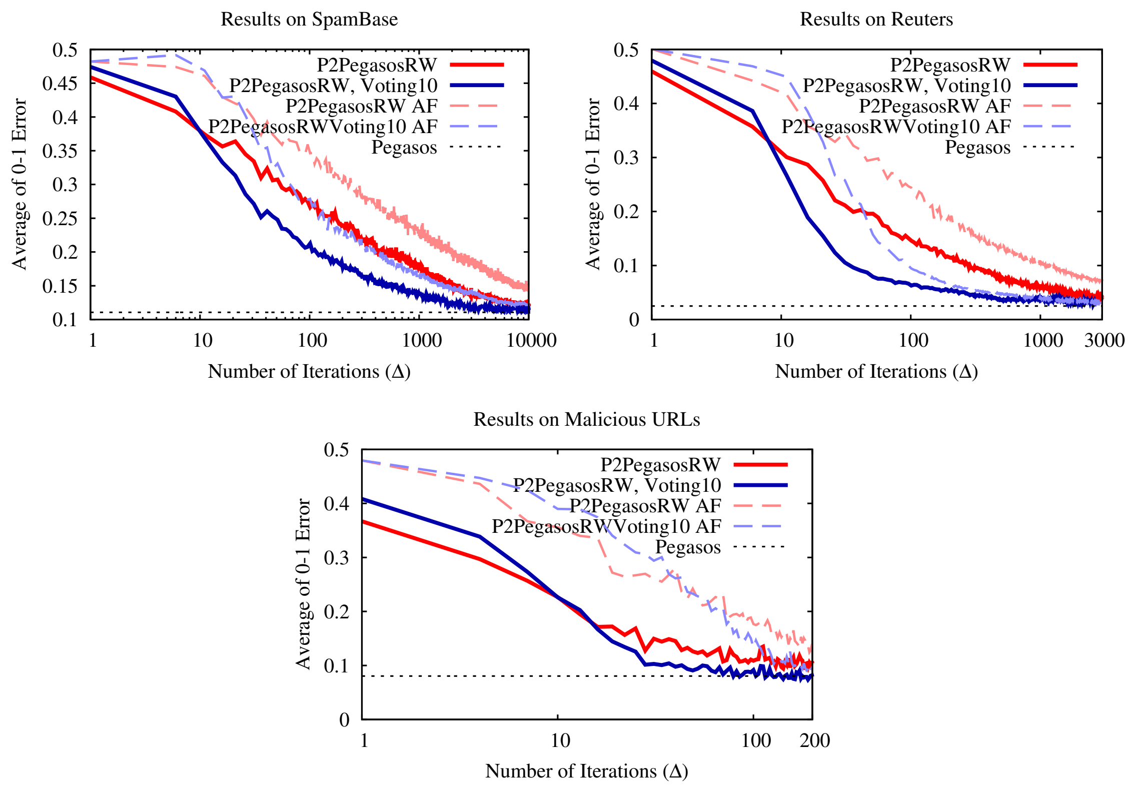

Figure 3.1. Experimental results without failure, with extreme failure (AF) and applying local voting.

3.6 Conclusions

We proposed a possible way of applying machine learning in a fully distributed environment, which is provided by a framework called GoLF. This framework allows us to model fully distributed data stored by nodes in a network, based on stochastic gradient search. The key point is that many models take random walks in the network and update themselves when reaching a node by its locally stored data. The model that visits a node is subsequently stored.

We presented several machine learning algorithms that can be incorporated into this framework and can be used for distributed data modeling. Furthermore, we presented an instantiation of the framework based on the Pegasos SVM algorithm. The results indicate that the algorithm is robust for extreme failures and that the local voting technique improves its performance.

The framework makes it possible to compute predictions locally at every node in the network at any point in time, yet the message complexity is acceptable: every node sends one model in each gossip cycle. The main characteristics that distinguish this approach from related work are the focus on fully distributed data and its modularity, generality and simplicity.

An important aspect of the approach is the support for privacy preservation, since data samples are not observed directly and never leave the nodes. The only feasible attack is the multiple forgery attack [79], where the local sample is guessed based on sending specifically crafted models to nodes and observing the result of the update step.

CHAPTER 4

Fully Distributed Boosting

In this chapter we present a well-known technique in GoLF, called boosting, which was published in our paper [52]. Boosting techniques have attracted growing attention in machine learning due to their outstanding performance in many practical applications. Here we develop a boosting algorithm, which proves the viability of gossip learning also for implementing state-of-the-art machine learning algorithms. In a nutshell, a boosting algorithm constructs a classifier in an incremental fashion by adding simple classifiers (that is, weak classifiers) to a pool. The weighted vote of the classifiers in the pool determines the final classification.

Our contributions are the following. First, to enable P2P boosting via gossip, we derive a purely online multi-class boosting algorithm, based on FilterBoost, which can be proven to minimize a certain negative log likelihood function. So this online boosting algorithm can be employed in GoLF. We also introduce efficient multi-class weak learners to be used by the online boosting algorithm. Second, we improve GoLF to make sure that the diversity of the models in the network is preserved. This makes it meaningful to spread the current best model in the network; a technique we propose to improve local prediction performance. We evaluate the robustness and the convergence speed of the algorithm empirically over three benchmark databases. We compare the algorithm with the sequential AdaBoost algorithm and we test its performance in a failure scenario involving message drop and delay, and node churn.

4.1 Background and Related Work

Here we consider the multi-class classification problem, where the labels are represented as a vector (see section 2.1). Furthermore, we will denote the index of the correct class by $\ell(\mathbf{x}_i)$. In classical multi-class classification the elements of $\mathbf{f}(\mathbf{x})$ are treated as posterior scores corresponding to the labels, so the predicted label is $\widehat{\ell}(\mathbf{x}) = \text{argmax}_{\ell=1,\dots,K} f_\ell(\mathbf{x})$ where $f_\ell(\mathbf{x})$ is the ℓth element of $\mathbf{f}(\mathbf{x})$. The function $\mathbf{f}$ is called the *model* of the data.

As mentioned before, in this chapter we focus on online boosting in GoLF. A few proposals for online boosting algorithms are known. An online version of AdaBoost [38] is introduced in [35] that requires a random subset from the training data for each boosting iteration, and the base learner is trained on this small sample of the data. The algorithm has to sample the data according to a non-uniform distribution making it inappropriate for pure online training. A gradient – based online algorithm is presented in [11], which is an extension of Friedman's gradient – based framework [39]. However, their approach is for binary classification, and it is not obvious how it can be extended to multi-class problems. Another notable online approach is Oza's online algorithm [90], whose starting point is AdaBoost.M1 [37]. However, AdaBoost.M1 requires the base learning algorithm to achieve 50% accuracy for any distribution over the training instances. This makes it impractical in multi-class classification since most of the weak learners used as a base learner do not satisfy this condition.

We also discuss work related to fully distributed P2P data mining in general. We note that we do not overview the extensive literature of parallel machine learning algorithms because they have a completely different underlying system model and motivation. We do not discuss those distributed machine learning approaches either that assume the availability of sufficient local data to build models locally (a survey can be found in [92]).

One notable and relevant research direction is gossip – based algorithms where convergence to global functions over fully distributed data is achieved through local communication. Perhaps the simplest example is gossip – based averaging [58, 63], where the gossip approach is extremely robust, scalable, and efficient. However, gossip algorithms support more sophisticated algorithms that compute more complex global functions. Examples include the EM algorithm [69], LDA [9] or PageRank [57]. Numerous other P2P machine learning algorithms have also been proposed, as in [77, 101]. A survey of many additional ideas can be found in [28]. This work builds on the Gossip Learning Framework (GoLF, see Chapter 3), which offers an abstraction to implement a wide range of machine learning algorithms.

4.2 Multi-Class Online FilterBoost

This section introduces our main contribution, a multi-class online boosting algorithm that can be applied in GoLF. We build on FILTERBOOST [22], where the main idea is to filter (sample) the training examples in each boosting iteration and to give the base learner only this smaller, filtered subset of the original training dataset, leading to fast base learning. The performance of the base classifier is also estimated on an additional random subset of the training set resulting in further improvement in speed.

Our formulation of the FILTERBOOST algorithm is given as Algorithm 4.10. This is not yet in a form to be applied in GoLF, but the transformation is trivial as discussed in Section 4.4. This fully online formulation is equivalent to FILTERBOOST, except that it handles multi-class problems as well. To achieve this, while ensuring that the algorithm can still be theoretically proven to converge, our key contribution is the derivation of a new weight formula calculated in line 22. First we introduce this formula, then we explain Algorithm 4.10 in more detail.

A boosting algorithm can be thought of as a minimization algorithm of an appropriately defined target function over the space of models. The target function is related to the classification error over the training dataset. The key idea is that we select an appropriate target function that will allow us to both derive an appropriate weight, as well as argue for convergence. Inspired by the logistic regression approach of [25], we will use the following negative log likelihood

Algorithm 4.10 FILTERBOOST$(\text{INIT}(), \text{UPDATE}(\cdot,\cdot,\cdot,\cdot), T, C)$

```
1: $\mathbf{f}^{(0)}(\mathbf{x}) \leftarrow 0$
2: for $t \leftarrow 1 \rightarrow T$ do
3:     $C_t \leftarrow C \log(t+1)$
4:     $\mathbf{h}^{(t)}(\cdot) \leftarrow \text{INIT}()$
5:     for $t' \leftarrow 1 \rightarrow C_t$ do                                  ▷ Online base learning
6:         $(\mathbf{x},\mathbf{y},\mathbf{w}) \leftarrow \text{FILTER}(\mathbf{f}^{(t-1)}(\cdot))$              ▷ Draw a weighted random instance
7:         $\mathbf{h}^{(t)}(\cdot) \leftarrow \text{UPDATE}(\mathbf{x},\mathbf{y},\mathbf{w},\mathbf{h}^{(t)}(\cdot))$
8:     end for
9:     $\gamma \leftarrow 0, W \leftarrow 0$
10:    for $t' \leftarrow 1 \rightarrow C_t$ do                          ▷ Estimate the edge on a filtered data
11:        $(\mathbf{x},\mathbf{y},\mathbf{w}) \leftarrow \text{FILTER}(\mathbf{f}^{(t-1)}(\cdot))$              ▷ Draw a weighted random instance
12:        $\gamma \leftarrow \gamma + \sum_\ell^K w_\ell h_\ell^{(t)}(\mathbf{x}) y_\ell, W \leftarrow W + \sum_\ell^K w_\ell$
13:    end for
14:    $\gamma \leftarrow \gamma / W$                                             ▷ Normalize the edge
15:    $\alpha^{(t)} \leftarrow \frac{1}{2} \log \frac{1+\gamma}{1-\gamma}$
16:    $\mathbf{f}^{(t)}(\cdot) = \mathbf{f}^{(t-1)}(\cdot) + \alpha^{(t)} \mathbf{h}^{(t)}(\cdot)$
17: end for
18: return $\mathbf{f}^{(T)}(\cdot) = \sum_{t=1}^T \alpha^{(t)} \mathbf{h}^{(t)}(\cdot)$
19: procedure FILTER($\mathbf{f}(\cdot)$)
20:    $(\mathbf{x},\mathbf{y}) \leftarrow \text{RANDOMINSTANCE}()$                         ▷ Draw random instance
21:    for $\ell \leftarrow 1 \rightarrow K$ do
22:        $w_\ell \leftarrow \frac{\exp\left(f_\ell(\mathbf{x}) - f_{\ell(\mathbf{x})}(\mathbf{x})\right)}{\sum_{\ell'=1}^K \exp\left(f_{\ell'}(\mathbf{x}) - f_{\ell(\mathbf{x})}(\mathbf{x})\right)}$
23:    end for
24:    return $(\mathbf{x},\mathbf{y},\mathbf{w})$
25: end procedure
```

function as our target function:

$$R_{\text{L}}(\mathbf{f}) = -\sum_{i=1}^n \ln \frac{\exp\left(f_{\ell(\mathbf{x}_i)}(\mathbf{x}_i)\right)}{\sum_{\ell'=1}^K \exp\left(f_{\ell'}(\mathbf{x}_i)\right)} = \sum_{i=1}^n \ln \left[1 + \sum_{\ell \neq \ell(\mathbf{x}_i)}^K \exp\left(f_\ell(\mathbf{x}_i) - f_{\ell(\mathbf{x}_i)}(\mathbf{x}_i)\right)\right] \tag{4.1}$$

Note that the FILTERBOOST algorithm returns a vector-valued classifier $\mathbf{f} : \mathbb{R}^d \rightarrow \mathbb{R}^K$.

FILTERBOOST builds the final classifier $\mathbf{f}$ as a weighted sum of *base classifiers* $\mathbf{h}^{(t)} : \mathbb{R}^d \rightarrow \{-1, +1\}^K$ returned by a *base learner* algorithm which has to be able to handle weighted training data. The class-related weight vector assigned to $\mathbf{x}_i$ in iteration t is denoted by $\mathbf{w}_i^{(t)}$ and its ℓth element is denoted by $w_{i,\ell}^{(t)}$. It can be shown that selecting $w_{i,\ell}^{(t)}$ so that it is proportional to the output of the current strong classifier

$$w_{i,\ell}^{(t)} = \frac{\exp\left(f_\ell^{(t)}(\mathbf{x}_i) - f_{\ell(\mathbf{x}_i)}^{(t)}(\mathbf{x}_i)\right)}{\sum_{\ell'=1}^{K} \exp\left(f_{\ell'}^{(t)}(\mathbf{x}_i) - f_{\ell(\mathbf{x}_i)}^{(t)}(\mathbf{x}_i)\right)}. \tag{4.2}$$

ensures that our target function in (4.1) will decrease in each boosting iteration. The proof is outlined in the Appendix of our paper [52].

The pseudocode of FILTERBOOST is shown in Algorithm 4.10. Here, the algorithm is implemented according to the practical suggestions given in [22]: first, the number of randomly selected instances is $C \log(t+1)$ in the tth iteration (where C is a constant parameter), and second, in the FILTER method the instances are first randomly selected then re-weighted based on their scores given by $\mathbf{f}^{(t)}(\cdot)$. Procedure INIT() initializes the parameters of the base classifier (line 4), and UPDATE$(\cdot,\cdot,\cdot,\cdot)$ updates (line 7) the parameter of the base classifier using the current training instance $\mathbf{x}$ given by FILTER$(\cdot)$. The input parameter T is the number of iterations, and C controls the number of instances used in one boosting iteration. $\alpha^{(t)}$ is the base coefficient, $\mathbf{h}^{(t)}(\cdot)$ is the vector-valued base classifier, and $\mathbf{f}^{(T)}(\cdot)$ is the final (strong) classifier. Procedure RANDOMINSTANCE() selects a random instance from the training data.

Let us point out that there is no need to store more than one training instance anywhere during execution. Second, the algorithm does not need any global information about the training data, such as the size, so this implementation can be readily applied in a pure online environment.

4.3 Multi-Class Online Base Learning

For the online version of FILTERBOOST, we need to propose online base learners as well. In FILTERBOOST, for theoretical reasons, the base classifiers are restricted to output discrete predictions in $\{-1, +1\}^K$ and, in addition, they have to minimize the weighted exponential loss

$$E(\mathbf{h},\mathbf{f}^{(t)}) = \sum_{i=1}^{n}\sum_{\ell=1}^{K} w_{i,\ell}^{(t)} \exp\left(-h_\ell(\mathbf{x}_i)y_{i,\ell}\right). \tag{4.3}$$

We follow this approach and, in addition, we build on our base learning framework [62] and assume that the base classifier $\mathbf{h}(\mathbf{x})$ is vector-valued and represented as $\mathbf{h}_\Theta(\mathbf{x}) = \operatorname{sign}(\mathbf{v}\varphi_\Theta(\mathbf{x}))$, parameterized by $\mathbf{v} \in \mathbb{R}^K$ (the *vote vector*), and $\varphi_\Theta(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}$, a *scalar* base classifier parameterized by Θ. The coordinate-wise sign function is defined as $\operatorname{sign} : \mathbb{R}^K \to \{-1,+1\}^K$. In this framework, learning consists of tuning Θ and $\mathbf{v}$ to minimize the weighted exponential loss (4.3).

Since it is hard to optimize the non-differentiable function $\mathbf{h}_\Theta$ even in batch mode, we take into account only $\widehat{\mathbf{h}}_\Theta(\mathbf{x}) = \mathbf{v}\varphi_\Theta(\mathbf{x})$. This approach is heuristic as it is hard to say anything about the relation between $E(\mathbf{h}_\Theta, \mathbf{f}^{(t)})$ and $E(\widehat{\mathbf{h}}_\Theta, \mathbf{f}^{(t)})$, but in practice this base learning approach performs quite well.

Since $\varphi_\Theta(\cdot)$ is differentiable, the stochastic gradient descent (SGD) [21] algorithm provides a convenient way to train the base learner in an online fashion. The SGD algorithm updates the parameters iteratively based on one training instance at a time. Let us denote $Q(\mathbf{x},\mathbf{y},\mathbf{w},\mathbf{v},\Theta) = \sum_{\ell=1}^{K} w_\ell \exp\left(-y_\ell v_\ell \varphi_\Theta(\mathbf{x})\right)$. Then the gradient based parameter update can be calculated as follows:

$$\Theta^{(t'+1)} \leftarrow \Theta^{(t')} + \gamma^{(t')}\nabla_\Theta Q(\mathbf{x},\mathbf{y},\mathbf{w},\mathbf{v},\Theta) \tag{4.4}$$

$$\mathbf{v}^{(t'+1)} \leftarrow \mathbf{v}^{(t')} + \gamma^{(t')}\nabla_{\mathbf{v}} Q(\mathbf{x},\mathbf{y},\mathbf{w},\mathbf{v},\Theta) \tag{4.5}$$

This update rule can be used in line 7 of FILTERBOOST to update the base classifier. A simple decision stump or ADALINE [114] can be easily accommodated to this multi-class base learning framework. In the following we derive the update rules for a decision stump, that is, a one-decision two-leaf decision tree having the form

$$\varphi_{j,b}(\mathbf{x}) = \begin{cases} 1 & \text{if } x^{(j)} \geq b, \\ -1 & \text{otherwise,} \end{cases} \tag{4.6}$$

where j is the index of the selected feature and b is the decision threshold. Since $\varphi_{j,b}(\mathbf{x})$ is not differentiable with respect to b, we decided to approximate it by the differentiable sigmoidal function, whose parameters can be tuned using SGD.

The sigmoidal function can be written as

$$s_{j,\theta}(\mathbf{x}) = s_{j,(c,d)}(\mathbf{x}) = \frac{1}{1+\exp\big(-cx^{(j)}-d\big)}.$$

where $\Theta = (c,d)$. And $\varphi_{j,b}(\cdot)$ can be approximated by $\varphi_{j,b}(\mathbf{x}) \approx 2s_{j,\theta}(\mathbf{x}) - 1$. Then the weighted exponential loss of this so-called *sigmoidal decision stump* for a single instance can be written as

$$Q_j = Q_j\left(\mathbf{x},\mathbf{y},\mathbf{w},\mathbf{v},\Theta\right) = \sum_{\ell=1}^{K} w_\ell \exp\left(-v_\ell\left(2s_{j,\theta}(\mathbf{x})-1\right)y_\ell\right)$$

and its partial derivatives are

$$\begin{aligned}
\frac{\partial Q_j}{\partial v_\ell} &= -\exp\left(-v_\ell\left(2s_{j,\theta}(\mathbf{x})-1\right)y_\ell\right)w_\ell\left(2s_{j,\theta}(\mathbf{x})-1\right)y_\ell \\
\frac{\partial Q_j}{\partial c} &= -2\sum_{\ell=1}^{K}\exp\left(-v_\ell\left(2s_{j,\theta}(\mathbf{x})-1\right)y_\ell\right)w_\ell v_\ell y_\ell x^{(j)} s_{j,\theta}(\mathbf{x})\left(1-s_{j,\theta}(\mathbf{x})\right) \\
\frac{\partial Q_j}{\partial d} &= -2\sum_{\ell=1}^{K}\exp\left(-v_\ell\left(2s_{j,\theta}(\mathbf{x})-1\right)y_\ell\right)w_\ell v_\ell y_\ell s_{j,\theta}(\mathbf{x})\left(1-s_{j,\theta}(\mathbf{x})\right)
\end{aligned}$$

The initial value of c and d were set to 1 and 0, respectively (line 4 of Algorithm 4.10).

So far, we implicitly assumed that the index of feature j is given. To choose j, we trained sigmoidal decision stumps in parallel for each feature and we estimated the edge of each of them using the sequential training data as $\widehat{\gamma}_j = \sum_{t'=1}^{C_t}\sum_{\ell=1}^{K} w_{t',\ell}y_{t',\ell}\mathrm{sign}\big(v_\ell^{(t')}\varphi_{j,\Theta_j^{(t')}}(\mathbf{x}_{t'})\big)$. Finally, we chose the feature with the highest edge estimate $j^* = \mathrm{argmax}_j\widehat{\gamma}_j$.

In every boosting iteration we also train a *constant learner* (also known as y-intercept) and use it if its edge is higher than the edge of the best decision stump we found. The output of the constant learner does not depend on the input vector $\mathbf{x}$, that is $\varphi(\cdot) \equiv 1$, in other words it returns the vote vector $\mathbf{v}$ itself. Thus only $\mathbf{v}$ has to be learnt but this can be done easily by calculating the classwise edge $v_\ell = \sum_{t'=1}^{C_t} w_{t',\ell}y_{t',\ell}$.

4.4 GoLF Boosting

In order to adapt Algorithm 4.10 to GoLF (Algorithm 3.3), we need to define the permanent state of the FilterBoost model class, and we need to provide an implementation of the updateModel method. This is rather straightforward: the model instance has to store the actual strong learner $\mathbf{f}^{(t)}$ as well as the state of the inner part of the two for loops in Algorithm 4.10 so that updateModel could simulate these loops every time a new sample is processed.

This way, every model that is performing a random walk is theoretically guaranteed to converge so long as we assume that peer sampling works perfectly. However, there is a catch. Since in each iteration some nodes will receive more than one model, while others will not receive any, and since the number of models in the network is kept constant if there is no failure (since in each iteration all the nodes send exactly one model) it is clear that the *diversity* of models will decrease. That is, some models get replicated, while others "die out". Introducing failure makes things a lot worse, since we can lose models due to message loss, delay, and churn as well, which speeds up homogenization. This is a problem, because diversity is important when we want to apply techniques such as combination or voting [88, 89]. Without diversity these important techniques are guaranteed not to be effective.

The effects of decreasing diversity are negligible during the timespan of a few gossip cycles, but a boosting algorithm needs a relatively large number of cycles to converge (which is not a problem, since the point of boosting is not speed, but classification quality). So we need to tackle the loss of diversity. We propose Algorithm 4.11 to deal with this problem.

This protocol works as follows. A node sends models in an active cycle (line 4) only in two cases: it sends the last received model if there was no incoming model until 10 active cycles (line 6), otherwise it sends all of the models received since the last cycle (line 14). If there is no failure, then this protocol is guaranteed to keep the diversity of models, since all the models in the network will perform independent random walks. Due to the Poisson distribution of the number of incoming models in one cycle, the probability of bottlenecks is diminishing, and for the same reason the probability that a node does not receive messages for 10

Algorithm 4.11 Diversity Preserving GoLF

```
1: $currentModel \leftarrow$ initModel()
2: $modelQueue$.add($currentModel$)
3: $counter \leftarrow 0$
4: **loop**
5:     wait($\Delta$)
6:     **if** $modelQueue$.isEmpty() **then**
7:         **if** $counter = 10$ **then**
8:             $p \leftarrow$ selectPeer()
9:             sendModel($p, currentModel$)
10:            $counter \leftarrow 0$
11:        **else**
12:            $counter \leftarrow counter + 1$
13:        **end if**
14:    **else**
15:        **for all** $m \in modelQueue$ **do**
16:            $p \leftarrow$ selectPeer()
17:            sendModel($p, m$)
18:            $modelQueue$.remove($m$)
19:        **end for**
20:        $counter \leftarrow 0$
21:    **end if**
22: **end loop**

23: **procedure** ONRECEIVEMODEL($m$)
24:     $m$.updateModel($x, y$)
25:     $modelQueue$.add($m$)
26:     $currentModel \leftarrow m$
27: **end procedure**
```

cycles is also practically negligible.

If the network experiences message drop failures or churn, then the number of models circulating in the network will converge to a smaller value due to the 10 cycle waiting time, and the diversity can also decrease, since after 10 cycles a model gets replicated in line 9. Interestingly, this is actually useful because if the diversity is low, it makes sense to circulate fewer models and to wait most of the time, since information is redundant anyway. Besides, with reliable communication channels that eliminate message drop (but still allow for delay), diversity can still be maintained.

Finally, note that if there is no failure, Algorithm 4.11 has the same total message complexity as Algorithm 3.3 except for the extremely rare messages sent in line 4. In case of failure, the message complexity decreases as a function of failure rate; however, the remaining random walks do not get slower relative to Algorithm 3.3, so the convergence rate remains the same on average, at least if no model-combination techniques are used.

4.5 Experimental Results

In our experiments we examined the performance of our proposed algorithm as a function of gossip cycles, which is about the same as the number of training samples seen by any particular model. To validate the algorithm, we compared

Table 4.1. The main properties of the data sets, and the prediction errors of the baseline algorithms.

	CTG	PenDigits	Segmentation
Training set size	1,701	7494	2100
Test set size	425	3,492	210
Number of features	21	16	19
Class labels	1325/233/143	10 classes (uniform)	7 classes (uniform)
AdaBoost (DS)	0.109347	0.060715	0.069048
FilterBoost (DS, C30)	0.094062	0.071657	0.062381

it with three baseline multi-class boosting algorithms, all using the same decision stump (DS) weak learner. The first one is the multi-class version of the well known AdaBoost [99] algorithm, the second one is the original FilterBoost [22] method implemented for a single processor, with the setting $C = 30$, and the third one is the online version of FILTERBOOST (Algorithm 4.10). We used three multi-class classification benchmark datasets to evaluate our method, namely the CTG, the PenDigits and the Segmentation databases. These were taken from the UCI repository [12] and have different size, number of features, class distributions and characteristics. The basic properties of the datasets can be found in Table 6.1.

In the P2P experiments we used the PeerSim [84] simulation environment to model message *delay*, *drop* and peer *churn*. We used two scenarios: a perfect network without any delay, drop or churn; and a scenario with heavy failure where the message delay was drawn uniformly at random from the interval $[\Delta; 10\Delta]$, a message was dropped with a probability of 0.5 and the online/offline session lengths of peers were modeled using a real P2P bittorrent trace [1]. As our performance metric, we applied the well known *0-1 error* (or error rate), which is the proportion of test instances that were incorrectly classified.

Figure 4.1 illustrates the effect of parameter C. Larger values result in slower convergence but better eventual performance. The setting $C = 30$ represents a good tradeoff in these datasets, so from now on we fix this value.

We compared our online boosting algorithm to baseline algorithms as can be seen in Figure 4.2 (left hand side). The figure shows that the algorithms converge to a similar error rate, which was expected. Moreover, our online FILTERBOOST

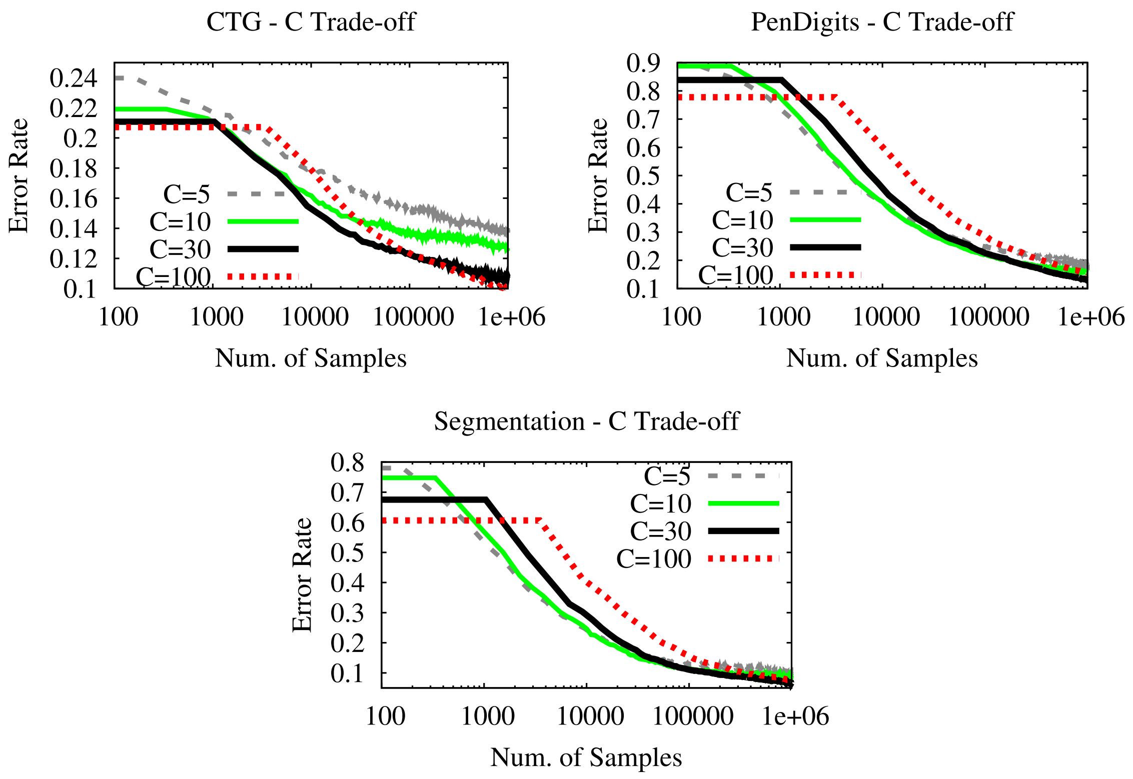

Figure 4.1. The effect of parameter C in online FILTERBOOST (Algorithm 4.10).

converges faster than the AdaBoost algorithm and it has almost the same convergence rate as that for the sequential FilterBoost method. Note that since two of these algorithms are not online, we had to approximate the number of (not necessarily different) training samples used in one boosting iteration. We used a lower bound to be conservative.

In our P2P evaluations of GOLF BOOSTING we used the mean error rate of 100 randomly selected nodes in the network to approximate the performance of the algorithm. Figure 4.2 (right hand side) shows that without failure the performance is very similar to that of our online FILTERBOOST algorithm. Moreover, in the extreme failure scenario, the algorithm still converges to the same error rate, although with a delay. This delay can be accounted for using a heuristic argument: since message delay in itself represents a slowdown of a factor of 5 on average, message drop and churn contributes approximately another factor of 2.

Finally, we demonstrate a novel way of exploiting model diversity (see Section 4.4): through gossip-based minimization one can spread the model with the

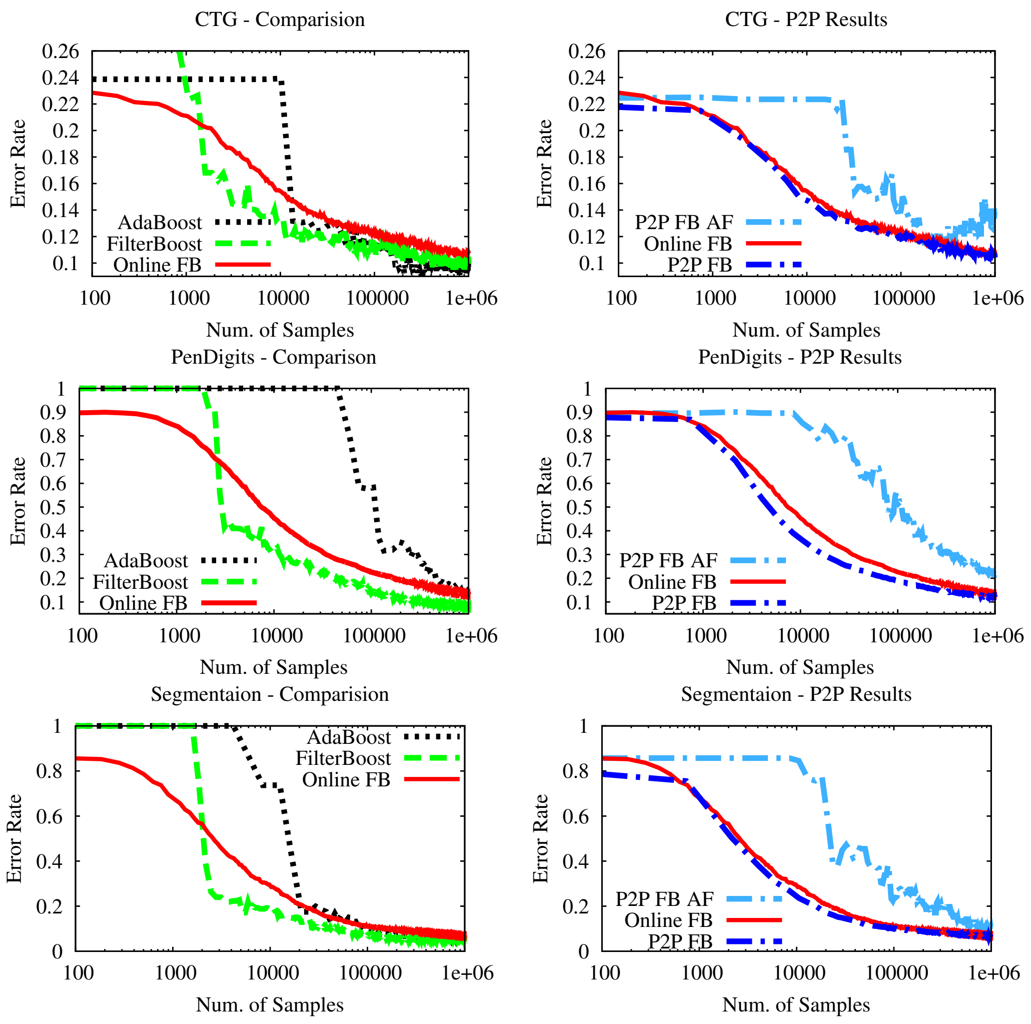

Figure 4.2. Comparison of boosting algorithms (left column) and P2P simulations (right column). FB and AF stand for FilterBoost and the "all failures" scenario, respectively.

best training performance, thus the best model can be made available to all nodes at all times. Figure 4.3 demonstrates this technique for different algorithms. We include results over the segmentation database only, the other two datasets produce similar results.

The top left plot shows results with GoLF Boosting. It can be seen that the best model based on training performance is not necessarily the best over the test set, but it is reasonably good, and results in a speedup of about a factor of 2. The top right plot belongs to the original GoLF implementation (Algorithm 3.3). Due to the complete lack of diversity, the best model's performance is almost

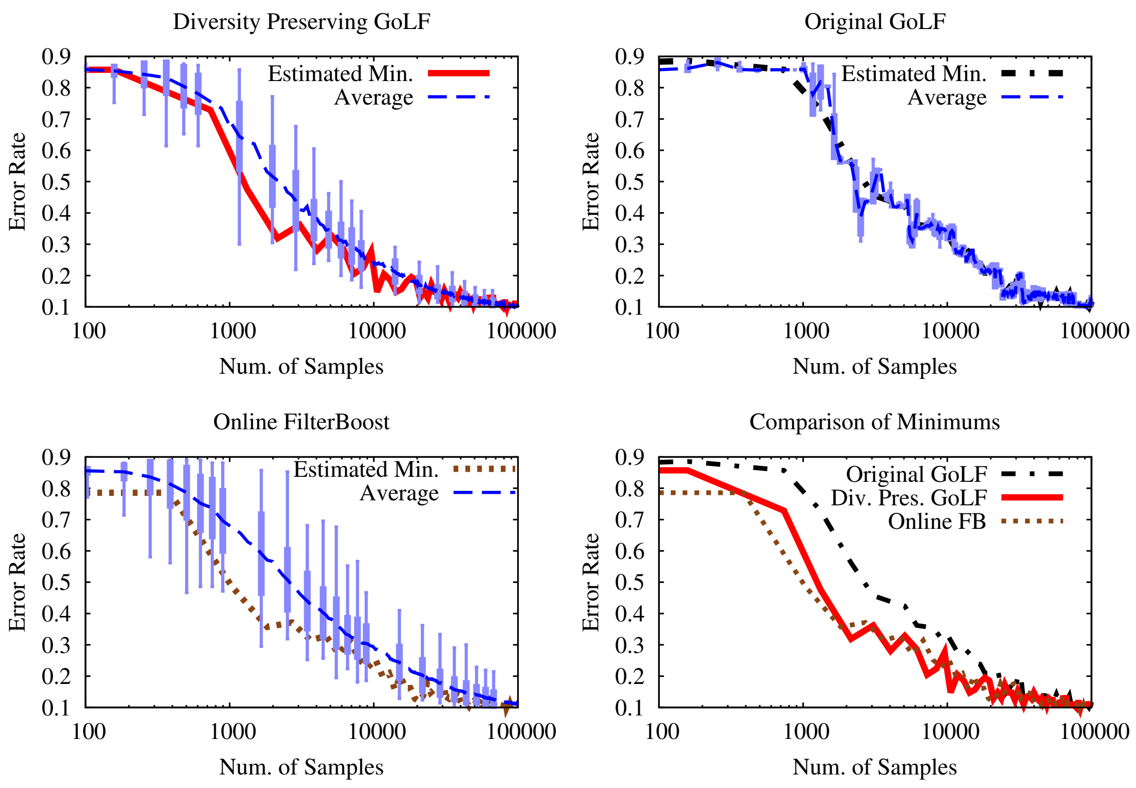

Figure 4.3. The improvement due to estimating the best model based on training performance. The Segmentation dataset is shown.

identical to the average one. The bottom left plot is a baseline experiment that represents the case with the maximal possible diversity, based on 100 completely independent runs of the online FILTERBOOST algorithm. Finally, the bottom right plot collects the most interesting curves from the other three plots allowing a better comparison.

4.6 Conclusions

We demonstrated that GOLF is suitable for the implementation of multi-class boosting. The significance of this result is that boosting is a state-of-the-art machine learning technique from the point of view of the quality of the learned models, which is now available in the P2P system model with fully distributed data. To achieve this, we proposed a modification of FILTERBOOST that allows it to learn multi-class models in a purely online fashion, and we proved theoretically that the resulting algorithm optimizes a suitably defined negative log likelihood measure. Our experimental results demonstrate the robustness of the method. We also identified the lack of model diversity as a potential problem with GoLF. We provided a solution that was demonstrated to be effective in preserving the difference between the best model and the average models; this allowed us to propose spreading the best model as a way to benefit from the large number of models in the network.

CHAPTER 5

Handling Concept Drift

Here we focus on the problem of *concept drift* [116] that occurs when the data patterns we wish to learn in the network change either continuously or suddenly. This can happen due to different reasons. For example, the set of users of an application can change, or external factors (such as the weather) can vary. This has an effect on how people react to, for example, traffic situations, it can influence what kind of movies they want to watch, it can affect their mobility patterns, and so on. People can also behave differently during their normal daily routine, or during a demonstration, for example. Such external factors cannot always be explicitly captured, and thus they can be considered a source of uncertainty and dynamism. The applied data mining algorithms have to follow these changes adaptively to provide up-to-date models at all times.

We are interested in scenarios, where data owned by a node cannot be moved outside the node. Currently the norm is that data is uploaded to data centers, where it is processed [15, 29, 76]. However, this practice raises serious privacy issues [26], and on a very large scale it can also become very expensive, which seriously reduces the set of potential applications.

In addition, we assume that data is distributed horizontally, that is, full data records (e.g. profiles, personal histories of sensor readings, etc) are stored at all nodes, but very few, perhaps only a single record is available at any given node. In addition, only a limited amount of new data can be sampled in order to follow concept drift. These assumptions are natural in the systems mentioned above, where, for example, a mobile device stores information only about its owner like user profiles, purchasing events, or mobility patterns. Besides, in applications, where people are even required to manually enter training data samples we cannot expect a huge amount of input at all nodes, especially if these samples correspond to rare events.

In Chapter 3 we have proposed the gossip learning framework (GoLF) for massively distributed data mining targeted to those environments that are characterized above [88, 89], where models of the data perform random walks in the network, while being improved using an online learning algorithm. However, one major limitation of GoLF was that it supported only one-shot algorithms, without taking adaptivity into account. This is a problem, because – as we argued above – in a realistic environment we are typically not able to take all relevant features into account when we learn data patterns, thus the validity of a pattern can change continuously or suddenly, in which case the learned data models need to be updated or refreshed. This can be achieved through a manual restart, but on a large scale, an automated method is necessary that allows the system to adapt to changes without human intervention. This would allow the system to scale both in terms of the size of the network, as well as the number of learning tasks supported simultaneously.

We propose two orthogonal approaches to follow concept drift in GoLF. The idea behind our first approach is that we manage the distribution of the lifetime of the models in the network, making sure that we have both young (and thus adaptive) models and old models at all times. This approach provides a very simple and efficient adaptive mechanism which makes it possible to seamlessly follow the changing data patterns. However, if change in the data patterns has to be detected explicitly, this solution is not suitable. For this reason, as our second contribution, we propose a mechanism that estimates and monitors the changes in the performance of the current data models and discards those ones that show a deteriorating performance.

Our contribution is twofold. First, we extend GoLF with components that allow it to deal with concept drift. With these changes, the algorithm can run indefinitely in a changing environment without any central control. Second, we perform a thorough experimental analysis. We compare the proposed algorithms with several baseline algorithms that idealize the main techniques for achieving adaptivity from related work. We show that in the domain where the number of independent samples available locally is low relative to the speed of drift, our solutions are superior to all the baselines and their performance approximates the theoretical maximum. We also demonstrate the fault tolerance of the method.

5.1 Related Work

We discuss the state-of-the-art related to concept drift in general, as well as in the area of P2P learning.

5.1.1 Non-Distributed Concept Drift Handling

A good overview of concept drift can be found in [116]. Many algorithms apply chunk based learning techniques [66, 104, 111], that is, they teach a new classifier when a new set of samples (a chunk) becomes available via the stream of samples. This approach could be suitable when the stream of samples produces a large number of samples relative to the speed of concept drift, that is, when the method can collect enough samples quickly enough to build an up-to-date classifier. Moreover, determining the chunk size is not easy, yet this parameter is crucial for the prediction performance.

One improvement of chunk based techniques is to detect drift, that is, to use some performance related measures to decide when to trigger the drift handling method [13, 40, 66]. The early approaches use a single model which is discarded when drift is detected and a new one is constructed immediately. Recently, ensemble learning has also been proposed [82]. In this case, when drift is detected, a new model is created but this new model is added to an ensemble pool that also contains older models. This pool is used to perform prediction, possibly involving a weighting mechanism as well.

Due to our system model, we are not able to collect chunks since there is not enough local data available. However, as we explain later, we will use a form of drift detection in which nodes cooperate with each other to detect drift.

5.1.2 Handling Concept Drift in Fully Distributed Environments

Learning in P2P systems is a growing area, some examples include [5, 6, 7, 28, 54, 77, 88, 101]. Very few works address issues related to concept drift in a P2P network. A fully distributed decision tree induction method was proposed by Bhaduri et al. [19]. The proposal involves drift detection, which triggers a tree update. The proposed solution is a distributed adaptive threshold detection algorithm that – although elegant – is a special purpose approach that does not generalize to arbitrary learning algorithms like our approaches do.

Another solution was proposed by Ang et al. [7]. This method implements the so-called RePCoDE framework which detects drift (reactive behavior) and simultaneously predicts it as well (proactive behavior). The basic learning mechanism is performed by chunk-based learning, but the models taught on previous data chunks are also kept and used during prediction (ensemble based aspect). As the extensive evaluations show, the proposed approach works well in various scenarios, although its communication cost is rather high, since it involves network wide model propagation. A number of heuristics are proposed to reduce this cost. In this paper we assume that chunk based approaches are not viable due to the lack of sufficient amounts of local data.

Our main contribution w.r.t. related work is to propose two *generic* methods to efficiently deal with the scenario, when samples arrive only very rarely at any given node, but in the overall network there are enough samples to learn high quality models.

5.2 Background

Here we briefly introduce an outline of the concept drift with the basic notations and our previous work on the field of distributed machine learning. In our presentation we used the supervised learning problem (see Chapter 2 for more details), where the drifting concepts occure.

5.2.1 Concept Drift

The distribution $\mathcal{D}$ mentioned above may change over time. For this reason we parameterize the distribution of the samples by the time t, that is, at time t a sample $(x, y) \in \mathbb{R}^d \times C$ comes from $\mathcal{D}_t$. This means that any learned prediction function f might become outdated if new samples are not used to update or replace it. The challenge is to design an *adaptive* algorithm that provides a good model f_t at any given time t. However, in some scenarios more might be required, for example, we might have to identify the time moment t^* when a sudden drift occurs. This problem is known as the *drift detection* problem.

Let us now elaborate on the types of concept drift that one can observe in the real world. The first type occurs when we are not using all the relevant and important features that are needed to provide accurate prediction. The reason can be that we do not know that the given feature is useful, or we might know it, but we might be unable to have access to the feature value. Examples include the age of users that is known to have an influence [96, 112] or sudden changes in weather conditions [113]. This type of drift can have very diverse time-scales from very quick (an environmental disaster) to very slow (the change of fashion) drift.

The second type of drift has to do with “arms race” situations when an adversary constantly changes strategy to overcome some security measures. The typical examples are the problem of spam filtering [30] and IT security [71]. In these cases, the models used to detect suspicious behavior or spam need to be updated constantly. This type of drift is typically rather slow.

5.2.2 Diversity Preserving GoLF

In GoLF the *diversity* of models will decrease, which is a problem, because diversity is important when we want to apply techniques such as combination or voting [88, 89]. More importantly, diversity is the key to achieve adaptation as well, the goal of this chapter. Algorithm 5.12 contains techniques to deal with this problem. These ideas to maintain diversity were first introduced in [52] and the detailed description can be seen in Chapter 4. Apart from the two commented lines that deal with concept drift (to be discussed later), the changes w.r.t. the original version are the mentioned diversity preservation techniques.

Finally, note that if there is no failure, Algorithm 5.12 has the same total message complexity as Algorithm 3.3 except for the extremely rare messages triggered by the timeout. In case of failure, the message complexity decreases as a function of failure rate; however, the remaining random walks do not get slower relative to Algorithm 3.3, so the convergence rate remains the same on average, at least if no model-combination techniques are used.

The discussion of the components of Algorithm 5.12 that deal with concept drift is delayed until Section 6.4.

5.3 Algorithms

In this section we present two alternative extensions to the original GoLF algorithm. The first one is an extremely simple approach that is independent of the drift pattern. Without trying to detect drift, we maintain a fixed age-distribution in order to keep the adaptivity and diversity of the models at a certain level. In the second approach, we explicitly detect drift via monitoring models and discarding the ones that perform badly. While this is a more complex approach (and therefore potentially more sensitive to the actual drift pattern), it is able to provide information about the actual drift in the system as well.

In Algorithm 5.12 we show the skeleton of the GoLF algorithm extended with two abstract methods to handle drift: INITDRIFTHANDLER and DRIFTHANDLER. The two approaches mentioned above are both implemented in this framework.

5.3.1 AdaGoLF: Maintaining a Fixed Age Distribution

It is well known that online algorithms must be less and less sensitive to new samples with time, otherwise they are unable to converge. However, this also means that after a certain point they are not able to adapt to a changing data distribution. To avoid models becoming too old, in our approach we achieve

Algorithm 5.12 GoLF with drift handling

```
1: $c \leftarrow 0$
2: $m \leftarrow$ initModel()
3: currentModel $\leftarrow$ initDriftHandler($m$)          ▷ drift handling: initialization
4: receivedModels.add(currentModel)
5: loop
6:     if receivedModels = $\emptyset$ then
7:         $c \leftarrow c + 1$
8:     end if
9:     if $c = 10$ then
10:        receivedModels.add(currentModel)
11:    end if
12:    for all $m \in$ receivedModels do
13:        $p \leftarrow$ selectPeer()
14:        send $m$ to $p$
15:        receivedModels.remove($m$)
16:        $c \leftarrow 0$
17:    end for
18:    wait($\Delta$)
19: end loop
20: procedure onReceiveModel($m$)
21:    $m \leftarrow$ driftHandler($m$)                     ▷ main drift handling method
22:    currentModel $\leftarrow$ updateModel($m$)
23:    receivedModels.add(currentModel)
24: end procedure
```

adaptivity by controlling the *lifetime distribution* of the models available in the network.

The implementation of age-based drift handling is shown in Algorithm 5.13. Note that this approach works *independently* of the learning algorithm applied in GoLF, and independently of the drift pattern as well. From now on, we will call the GoLF framework extended with age-based drift handling ADAGOLF.

The algorithm works by adding a new time-to-live (TTL) field to each model. When a new model is created, this field is initialized (at line 2 of Algorithm 5.13) to a value generated from a predefined probability distribution that we call the Model Lifetime Distribution (MLD). The age of the model increases with each hop. When the model age reaches the TTL value, the model is discarded and a

Algorithm 5.13 AdaGoLF

```
1: procedure INITDRIFTHANDLER($m$)
2:     $m.TTL \leftarrow$ generateTTL()
3:     return $m$
4: end procedure
5: procedure DRIFTHANDLER($m$)
6:     $m.age \leftarrow m.age + 1$
7:     if $m.age = m.TTL$ then
8:         $m \leftarrow$ initModel()
9:         $m \leftarrow$ initDriftHandler($m$)
10:    end if
11:    return $m$
12: end procedure
```

new one is created (at line 8 of Algorithm 5.13) with a newly generated, independent TTL value (at line 9 of Algorithm 5.13).

The MLD can be selected arbitrarily by a particular implementation to achieve optimal adaptivity. If the drift pattern is not known, then the MLD should have a reasonably heavy tail, so that we always have old models in the system as well as new ones. Such distributions are more robust to the speed and the pattern of concept drift given that a wide range of model ages are always available.

We would like to characterize the distribution of the age of the models in the network at some time point t, given the MLD. Consider a sequence of models $m_1, m_2, \ldots$ according to a random walk where m_i is removed and m_{i+1} is started by method DRIFTHANDLER. The *lifetime sequence* of these models $m_1.TTL$, $m_2.TTL, \ldots$ forms a *renewal process*. Let the age of the model that is "alive" at time t be the random variable A_t; we are interested in the distribution of A_t. More formally, let S_t be the birth time of the model alive at time t:

$$S_t = \max_k \{V_k : V_k = \sum_{i=1}^{k} m_i.TTL \text{ and } V_k < t\}, \tag{5.1}$$

in which case $A_t = t - S_t$. Applying results from renewal theory [44] (the renewal equation and the expectation equation) we get the expected model age and its variance as the time tends to infinity:

$$\begin{aligned}
\mathbb{E}(A_t) &\xrightarrow{t\to\infty} \frac{\mathbb{E}(X^2)}{2\mathbb{E}(X)} \\
\mathbb{D}^2(A_t) &\xrightarrow{t\to\infty} \frac{\mathbb{E}(X^3)}{3\mathbb{E}(X)} - \left(\frac{\mathbb{E}(X^2)}{2\mathbb{E}(X)}\right)^2,
\end{aligned} \tag{5.2}$$

where random variable X is from the MLD.

We selected the lognormal distribution as our MLD with parameters $\mu = 8$ and $\sigma^2 = 0.5$, which gives us an expected age of $\mathbb{E}(A_t) \approx 3155$, and $\mathbb{D}(A_t) \approx 3454$. This distribution has a reasonably long tail, so we are guaranteed to have old as well as new models at all times. For this reason we expect this distribution to perform well in a wide range of drift scenarios.

5.3.2 CDDGoLF: Detecting Concept Drift

We propose Algorithm 5.14 for detecting drift explicitly. Note that the algorithm is still independent of the applied learning algorithm so it can be used along with any GoLF implementation.

We extended the models with a queue of a bounded size, called *history*, that stores performance related data based on the previously seen examples. The main idea is that – when a node receives a model – we can use the sample stored at the node for evaluating the model before we use the sample to actually update the model. Storing these evaluations in the history, we can detect the trend of the performance of the model, in particular, we can detect whether the performance is decreasing.

Let us explain the algorithm in more detail. Procedure DRIFTHANDLER uses the locally stored sample as a test sample and evaluates the model (line 6 of Algorithm 5.14). Then it updates the model history by storing an error score (value 1 if the predicted and real class labels are different; 0 otherwise) in line 7 of Algorithm 5.14. Applying this technique we can accumulate a bounded size series of independent error scores in the history. We have to make two important observations about the history. First, the error sequence stored in the history is biased in the sense that the model we measure is continuously changing while we collect

Algorithm 5.14 CDDGoLF

```
1: procedure InitDriftHandler($m$)
2:     $m.history \leftarrow ()$
3:     return $m$
4: end procedure
5: procedure DriftHandler($m$)
6:     $\hat{y} \leftarrow m.\text{predict}(x)$                    ▷ $(x, y)$ is the local sample
7:     $m.history.\text{add}(\hat{y} = y\ ?\ 0 : 1)$            ▷ history update
8:     if driftOccurred($m$) then
9:         $m \leftarrow$ initModel()
10:        $m \leftarrow$ initDriftHandler($m$)
11:    end if
12:    return $m$
13: end procedure
14: procedure DriftOccurred($m$)
15:    $errorRates \leftarrow$ smooth($m.history$)
16:    $slope \leftarrow$ linearReg($errorRates$)
17:    if rand([0; 1]) < $\sigma_{c,d}(slope)$ then
18:        return $true$
19:    end if
20:    return $false$
21: end procedure
```

the error scores. In other words, this is merely a heuristic approach to characterize the performance trend. Second, we have to use a bounded size queue to keep the message size constant, since the history is a part of the model, so it is sent through the network each time we pass the model on. In our case we used a history size of 100.

After updating the history, method DRIFTOCCURRED is used to decide whether a concept drift occurred (line 8 of Algorithm 5.14). In this method we first perform a preprocessing step (line 15 of Algorithm 5.14) by applying a sliding window-based averaging with window size *history*.size()/2 on the raw data. This step is necessary for noise reduction reasons. Second, we apply linear regression [83] on the smoothed error rates (line 16 of Algorithm 5.14) to extract the trend in the performance history using the slope of the fitted linear approximation. Finally, we convert the slope value into a drift probability by applying a sigmoid function mapping it onto the interval $(0, 1)$. We then issue a concept drift alert with this probability. The sigmoid function we used is $\sigma_{c,d}(x) = (1 + e^{-c(x-d)})^{-1}$, with parameters $c = 20$ and $d = 0.5$. These parameters were not fine-tuned (noting that $d = 0.5$ is needed for a symmetric mapping).

The geometrical interpretation is that if the slope is negative or 0, then the model in question is still improving (learning phase) or stable (converged phase), respectively. Otherwise, if the slope is positive, then the performance of the model is decreasing, which is most likely due to concept drift.

When drift occurs (i.e. method DRIFTOCCURRED returns *true*), we reinitialize the model, that is, we discard the old one and start a fresh one.

5.3.3 The Learner Component

So far we have discussed only abstract algorithms that were independent of the actual machine learning algorithm that is implemented in GoLF. However, to evaluate our proposals, we need to implement an actual learning algorithm. In this paper, we opted for logistic regression. In our previous work we have proposed support vector machine (SVM) and boosting implementations as well [52, 89], but any online learning algorithm is suitable that has a constant or slowly growing model size. In Chapter 3 we presented some useful machine learning algorithms.

5.3.4 Communication Complexity

The expected communication cost for a single node in a period of Δ time (one cycle) is at most two. To see this, consider that there are M models in the entire network, and there are $N \geq M$ nodes (with $N = M$ if there is no message drop and no churn), and that every model performs a random walk with exactly one hop in each cycle. Since we assume a good quality peer sampling service, the number of incoming messages will follow a Poisson distribution with $\lambda = 1$ if $M = N$, and less if $M < N$. Since each incoming message also generates an outgoing message, the overall number of messages will be twice the number of the incoming ones.

The space complexity of a model, which directly determines message size, strongly depends on the selected learning algorithm, as well as the dimensionality of the data samples. In the case of a linear model we apply in this section, the size of a model is the same as the size of a data sample. In addition, a model might also contain drift handling information such as the age value or a performance history of a bounded size.

5.4 Experimental Setup

5.4.1 Drift Dynamics and Drift Types

Our algorithm is not the optimal choice in all possible concept drift scenarios, however, in certain important cases we will show it to be the most favorable option. To be able to characterize the different drift scenarios, let us first identify a few key features of drift.

As of dynamics, there are two important properties that characterize an environment involving concept drift: the *speed of drift*, and the *sampling rate*. The speed of drift defines how much the underlying concept changes within a unit time interval. The sampling rate defines how many new *independent* samples become available within a unit time interval. We need to stress that only independent samples count in this metric, that is, samples that are drawn from the underlying distribution independently at random.

A third speed-related parameter is the cycle length Δ of GoLF. However, these three parameters can be considered redundant, since in the range of reasonable cycle lengths Δ (where Δ is significantly greater than message transmission time) we can always choose a Δ that keeps drift speed (or sampling rate) constant. For this reason, we will choose Δ as the unit of time, and we will define drift speed and sampling rate in terms of Δ, leaving us with two remaining free parameters.

However, in this paper we do not investigate the speed of drift independently, since the interesting scenarios are differentiated more by the *ratio* of drift speed and sampling rate. If drift is too fast relative to the sampling rate, then there is no chance to learn a reasonable model with any method. If drift is too slow relative to the sampling rate, then the problem is not very challenging, since even the simplest baselines can achieve a very good performance [116].

The type of the drift is another important property. In our scenarios drift can be *incremental* or *sudden* [116]. The actual drift types are described later in this section.

5.4.2 Baseline Algorithms

We selected our baseline algorithms so as to represent the most typical approaches from related work with a simpler, but optimistic version, which is guaranteed to perform better by construction than the corresponding published algorithm.

Our simplest baseline is LOCAL, where each node simply collects its local samples during a cycle and builds a model based on this sample set at the end of each cycle. If a node does not observe any training samples during a cycle, then the previous model is used for prediction. This solution involves no communication, but with low sample rates it performs poorly.

CACHEBASED is a more sophisticated baseline which uses a limited size memory, a FIFO queue (called cache) in which it collects samples. For maximal fairness, the memory size was optimized during preliminary experiments, and was set to 100. Note that our test datasets are learnable from 100 samples very well. Due to the optimized cache size, this baseline represents the chunk based as well as the trigger based approaches mentioned in Section 5.1. The cache size can be considered an optimized chunk size, which is the optimal solution of the trigger based approaches as well assuming continuous drift. Thus the result of this baseline can be considered as an upper bound of the performance of the local chunk and trigger based approaches. This baseline still uses no communication, but it performs better than LOCAL.

The baselines VOTE and CACHEDVOTE are natural extensions of the previous baselines: they use collaboration within the network. All the nodes send their models they create at the end of the cycle to every other node, and the prediction is performed by voting. These are powerful baselines in terms of prediction performance, although their communication costs are extremely large due to a full broadcast of all the models in each cycle. It is easy to see that these

Table 5.1. The main properties of the baseline and the adaptive algorithms.

Name	Computational complexity of learning / cycle	# models used for prediction	Communication complexity / cycle
LOCAL	O(chunkSize)	1	0
CACHEBASED	O(sampleCacheSize)	1	0
VOTE	O(chunkSize)	N	$O(N)$
CACHEDVOTE	O(sampleCacheSize)	N	$O(N)$
VOTE WD	O(chunkSize)	modelCacheSize	$O(1)$
CACHEDVOTE WD	O(sampleCacheSize)	modelCacheSize	$O(1)$
GLOBAL	O(chunkSize*N)	1	$O(N)$
ADAGOLF	$O(1)$	1	$O(1)$
CDDGOLF	$O(1)$	1	$O(1)$

voting-based baselines represent the ensemble based approaches mentioned in Section 5.1, hence the performance of these idealized variants can be considered an upper bound of the result of the ensemble based approaches.

VOTE WD and CACHEDVOTE WD are the communication-effective versions of VOTE and CACHEDVOTE (WITH DELAY), respectively. They work exactly the same way like their ancestors but the model spreading process is slowed down: in each cycle each node sends its model to exactly one neighbor. These models are collected in a FIFO queue of a fixed size (the model cache) and prediction is based on the voting of the models in the model cache. We set the model cache size to be 100, similarly to the sample cache described above.

The last baseline called GLOBAL is an algorithm that can observe all the samples observed in the network in a cycle and can build a model based on them. The implementation of this algorithm is infeasible and its result can be considered as a theoretical upper bound on the performance of the distributed baselines and adaptive algorithms.

The complexity of these algorithms is summarized in Table 5.1. The network size is denoted by N, CHUNKSIZE is the number of samples observed by a node in a cycle, which depends on the sampling rate. The last two lines show the same properties for our algorithms.

5.4.3 Data Sets

In our evaluations we used synthetically generated as well as real world data sets. In both cases we modeled drift by changing the labeling of the data set. That is, drift itself was synthetic even in the case of the real world database. The nodes can get random samples according to the given sampling rate from a training pool.

The synthetic database was generated by drawing uniform random points from the d-dimensional uniform hypercube. The labeling of these points is defined by a hyperplane that naturally divides the examples into a positive and negative subset. Drift is modeled by moving the hyperplane periodically over time [55]. A hyperplane at time t is defined by its normal

$$w_t = (1 - \alpha_t)w_s + \alpha_t w_d, \tag{5.3}$$

where $w_t, w_s, w_d \in \mathbb{R}^d$, $0 \leq \alpha_t \leq 1$, and w_s and w_d are the source and the destination hyperplanes, respectively, which are orthogonal to each other. The normal w_s is generated at random with all coordinates drawn from $[0, 1]$ uniformly, and $w_d = (1, \ldots, 1)^T - w_s$.

In the case of incremental drift the hyperplane is moved smoothly back and forth between w_s and w_d. This can formally be defined as

$$\alpha_t = \begin{cases} 1 - (tv - \lfloor tv \rfloor) & \text{if } \lfloor tv \rfloor \mod 2 = 1 \\ tv - \lfloor tv \rfloor & \text{otherwise} \end{cases} \tag{5.4}$$

where v is the speed of drift. When sudden drift was modeled, we used the same dynamics for α_t but rounded it to implement discontinuity: $\alpha'_t = \text{round}(\alpha_t)$. This results in switching back and forth between w_s and w_d with a period of $2/v$ time units.

We used one real world data set, namely the Image Segmentation [12] data set from the UCI Machine Learning Repository. The database has 19 real valued features and 7 class labels. The class label ratio is balanced. Originally this data set does not support the evaluation of concept drift. We implemented a mechanism proposed by [31, 110] to add synthetic drift to this data set. This simple mechanism consists of rotating the class labels periodically. This results in a variant of sudden drift.

5.4.4 Evaluation Metrics

For performing evaluations we split all the databases into a training and an evaluation set. The proportion of this splitting was 80/20% (training/evaluation) except in the case of the real data set where the training/evaluation split was provided by the owners of the database. In all cases the splitting was performed before the simulations since all the evaluation sets are obviously independent from the training sets.

Our main evaluation metric is prediction error. In the case of AdaGoLF and CDDGoLF, we track the misclassification ratio over the test set of 100 randomly selected peers. The misclassification ratio of a model is simply the number of the misclassified test examples divided by the number of all test examples, which is also called the 0-1 error.

5.4.5 Simulation Scenarios

The experiments were performed using the event-based engine of PeerSim [84]. Apart from experiments involving no failure, we model the effect of message drop, message delay, and churn. In all the failure scenarios we modeled realistic churn, that is, the nodes were allowed to join and leave the network. While offline, the nodes retain their state. The length of the online sessions was drawn at random from a lognormal distribution. The parameters of this distribution were given by the maximum likelihood estimation over a private BitTorrent trace called the FileList.org collected by Delft University of Technology [1]. For example, the average session length is around 5 hours. Note that in mobile phone networks, one can expect similarly long, or even longer sessions on average.

So far our unit of time was Δ (the cycle length), but here we need to specify Δ in order to be able to model churn in simulation. A normal value of Δ in an implementation would be around 10 seconds. However, with an average session length of 5 hours, this would result in practically no churn from our point of view, since convergence occurs at a much faster timescale. So, in order to push our algorithms to the limit, we set Δ to be the average session length, which results in an extreme churn scenario with nodes joining and leaving very frequently. In most of the experiments, except those in Section 5.5.6, we set the offline session lengths so that at any moment in time 10% of the nodes are offline. In Section 5.5.6 we experiment also with scenarios where 50% or 80% of the peers are offline.

In the scenario we call *AF mild* (all failures mild), message drop probability was set to be 0.2, and message delay is modeled as a uniform random delay from the interval $[\Delta, 2\Delta]$, where Δ is the cycle length. Moreover we applied a very heavy failure scenario as well where the message drop probability was 0.5 and message delay was uniform random from $[\Delta, 10\Delta]$. We refer to this scenario as *AF hard*.

The default value for the sampling rate parameter was $1/\Delta$, and the default for network size is $N = 100$. Both of these values are explored in our experimental study.

5.5 Experimental Results

5.5.1 Adaptivity

First we illustrate the problem that is caused by the lack of adaptivity. Online learners exhibit a burn in effect when run for a long time, as demonstrated by Figure 5.1, where we present the prediction error (averaged over the network) of GoLF (without drift handling), AdaGoLF, and CDDGoLF as a function of time. We can see that AdaGoLF and CDDGoLF show no burn in effect (these two algorithms have very similar performance producing mostly overlapping curves).

5.5.2 Drift Detection

We start with evaluating the drift detection heuristic of CDDGoLF. Here we focus on the sudden drift scenarios. The results are shown in Figure 5.2, where the rows represent different databases and the columns represent different network sizes. In each figure we present the average 0-1 error (Error), and the cumulative proportion of drift alarms over all the models that are processed in the network (Cum. det.) as a function of time. This metric is computed by calculating the proportion of those calls to driftHandler that result in an alarm in each cycle,

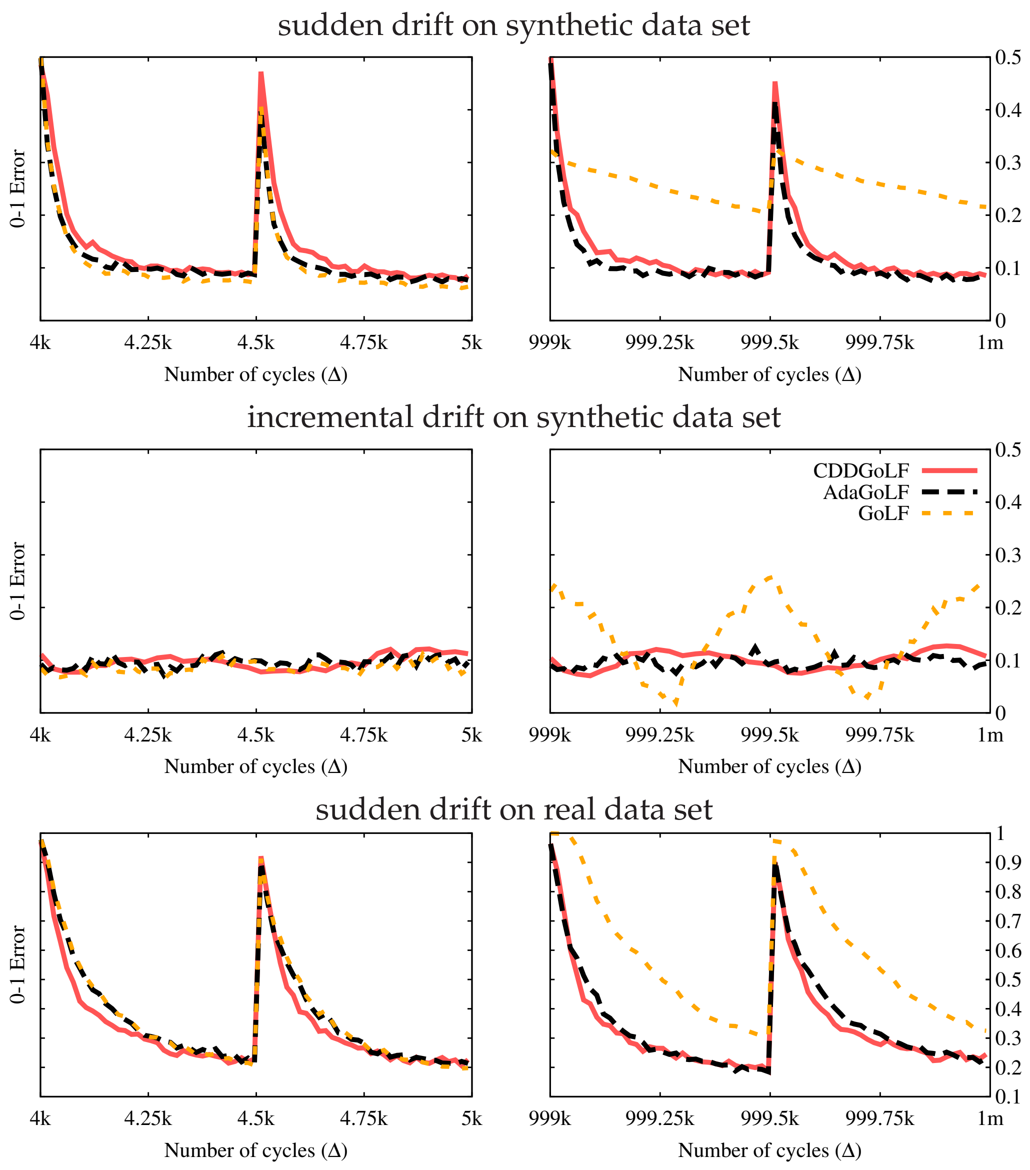

Figure 5.1. The burn in effect as a motivation for adaptivity.

and summing these proportions over the cycles up to the next underlying sudden drift, when we reset the sum to zero.

In the case of the real world database, the speed of drift detection is extremely fast: the cumulative detection curve increases very sharply when drift occurs. In the case of the synthetic database, the detection is slower. A possible reason is that drift is less dramatic (note that in the synthetic case the maximal error is 0.5,

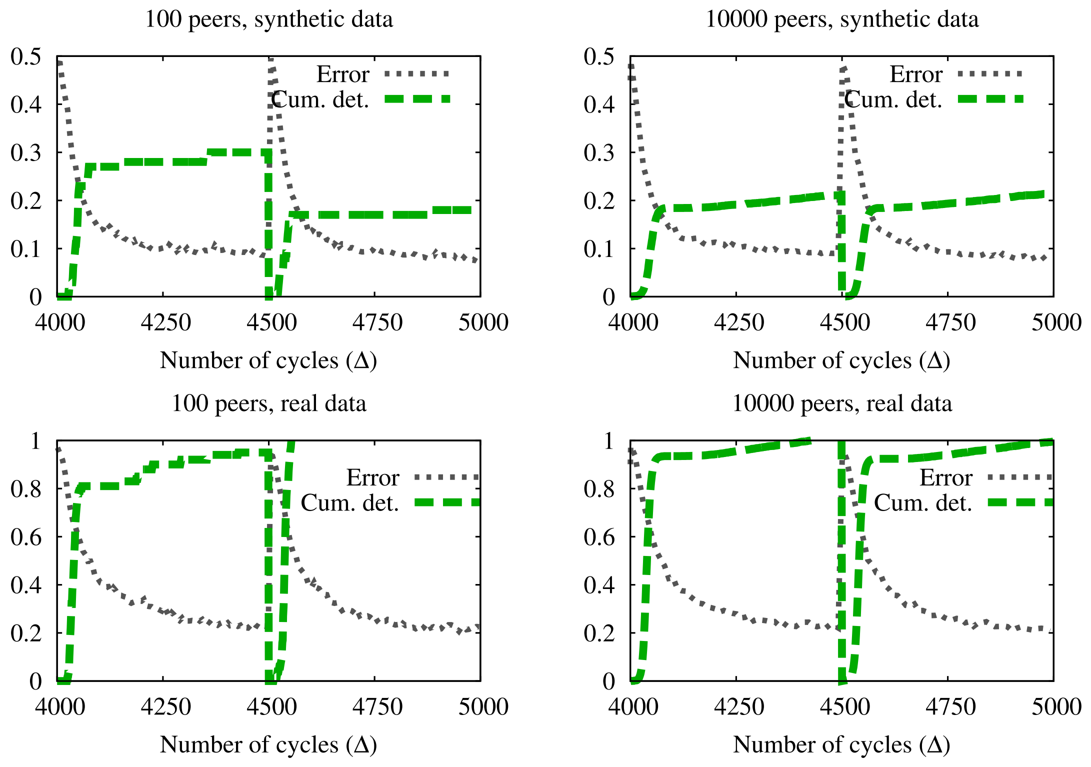

Figure 5.2. The drift detection and the classification performance of the proposed method on synthetic and real datasets.

while in the real database it is 1) since half of the labels remain correct after the sudden change. Based on both databases, these results indicate that CDDGoLF detects the drift accurately and quickly.

Let us now turn to the discussion of the effect of network size: We cannot see any significant difference in the error rate between the simulations for sizes 100 and 10,000 (left and right column, respectively). This observation implies the scalability of CDDGoLF. This is not a surprise, since GoLF actually only benefits from scale because there are more independent learning samples and there is a higher diversity of models.

5.5.3 The Effect of Sampling Rate

We now turn to the problem of identifying those scenarios, in which GoLF is preferable over the alternative approaches for dealing with concept drift. We

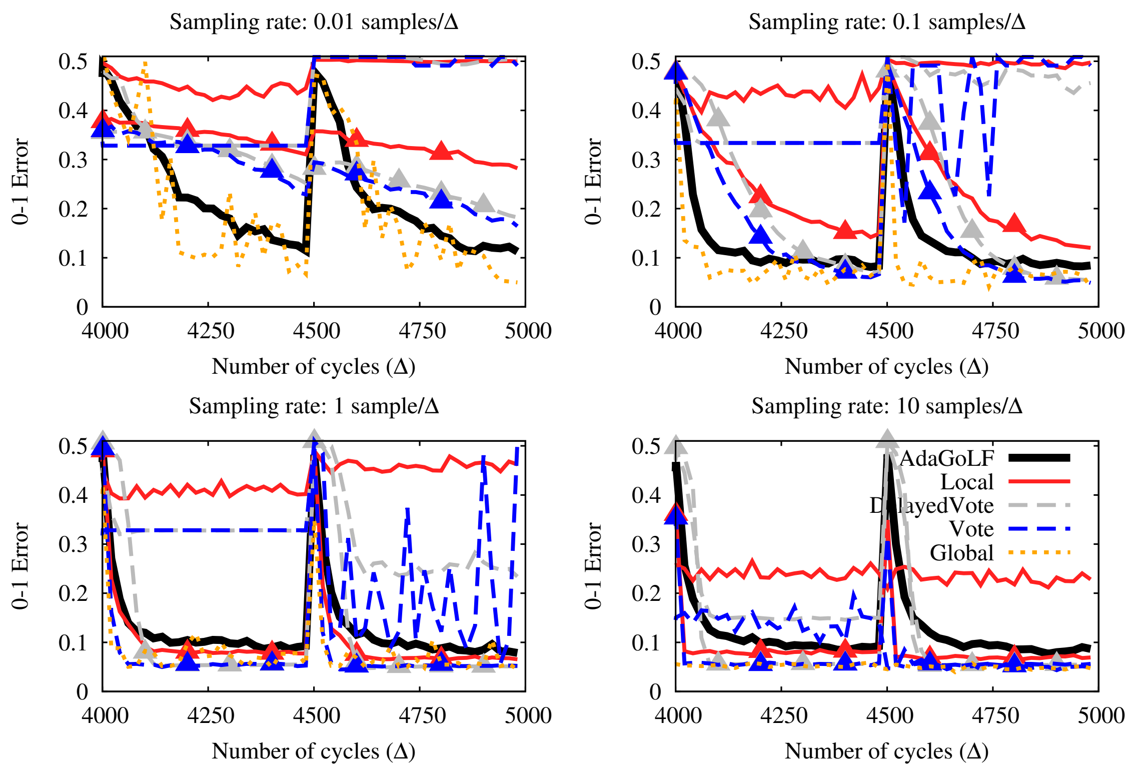

Figure 5.3. The effect of sampling rate under sudden drift (the lines marked with △ belong to cache based baselines).

focus on sampling rate, as described in Section 5.4. We performed evaluations using all our database and drift-type configurations (synthetic-sudden, synthetic-incremental and real world-sudden). The results are shown in Figures 5.3, 5.4 and 5.5. In each result set we selected four distinct sampling rates: 0.01, 0.1, 1, 10 samples per cycle.

We have performed these experiments both with ADAGOLF and CDDGOLF. Since these two algorithms show almost identical performance, from now on we include only ADAGOLF in the discussion.

When the sampling rate is 0.01, all the nodes receive only a single new sample on average in every 100 iterations. While the baseline methods build models only based on local samples, ADAGOLF can take advantage of many more samples due to the models performing a random walk. This allows ADAGOLF to approximate the performance of GLOBAL that has the best possible performance

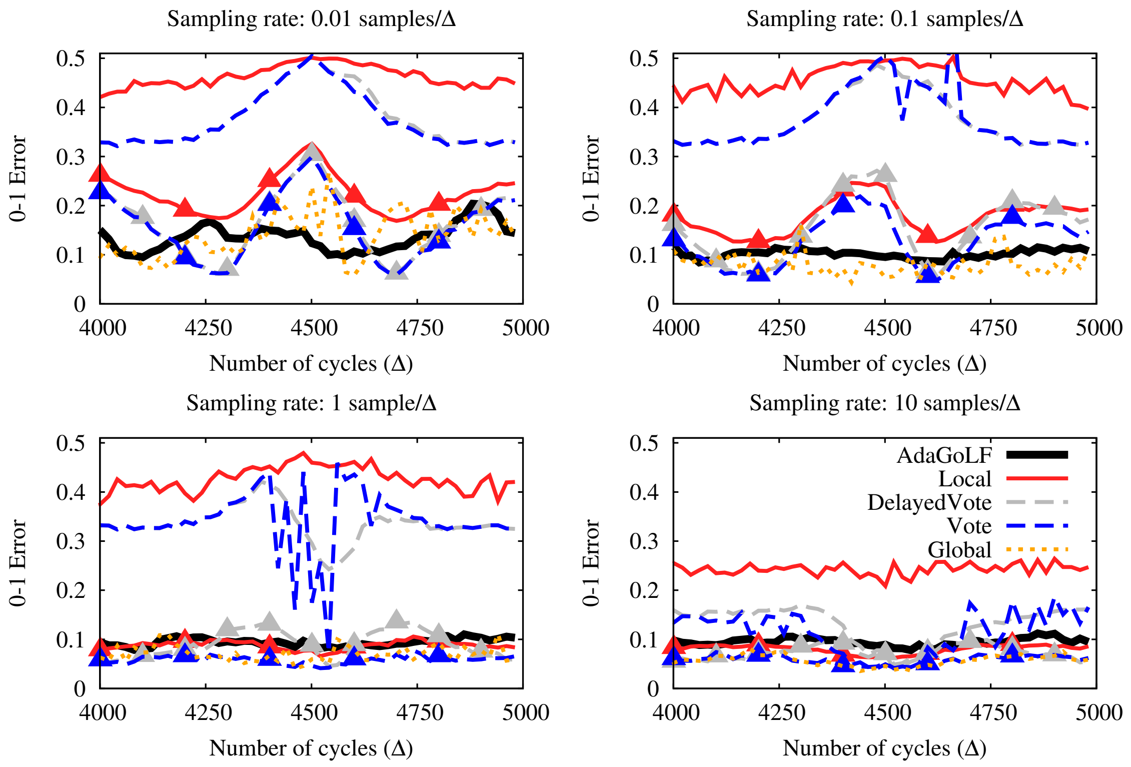

Figure 5.4. The effect of sampling rate under incremental drift (the lines marked with △ belong to cache based baselines).

by construction. Increasing the sampling rate results in a gradually decreasing difference between the baseline algorithms and ADAGOLF. In fact, with high sampling rates, ADAGOLF is outperformed by most of the baselines in the case of the sudden change scenarios.

We need to note here that in the present version of the algorithm all models use exactly one sample for the update in each hop, even if more samples are available. This means that there are lots of possibilities to enhance ADAGOLF to deal with high sample rates better. Nevertheless, ADAGOLF is clearly the best option if the sampling rate is low.

In the incremental drift scenario, we should mention the remarkable stability of ADAGOLF under each sampling rate. Here, ADAGOLF remains competitive even in the highest sampling rate scenario. This is rather interesting, given that ADAGOLF ignores most of the samples in that case, as mentioned before.

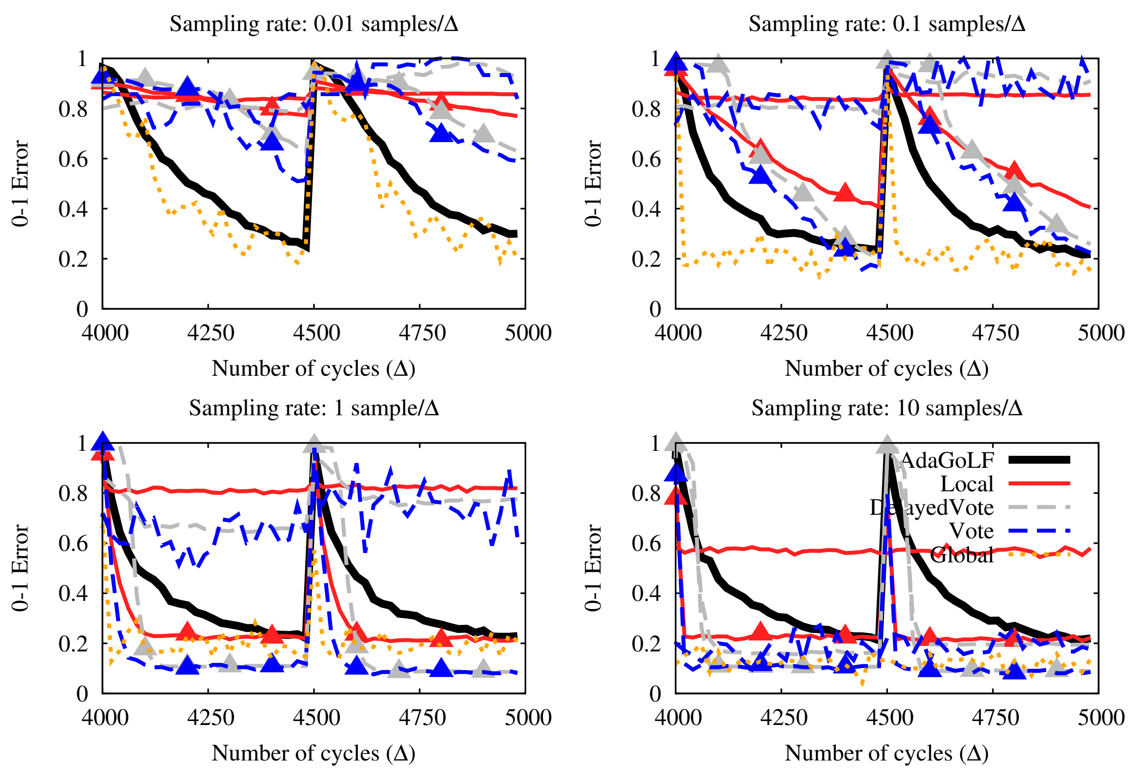

Figure 5.5. The effect of sampling rate over the real database (the lines marked with △ belong to cache based baselines).

We would like to stress again, that the scenario in which there are relatively few local *independent* samples relative to the speed of drift is practically very relevant. In the majority of important applications in P2P or mobile networks there is practically one sample throughout the entire participation of a certain user (in which case drift originates mostly from the changing membership). For example, when the data consists of relatively stable personal preferences, stable mobility patterns (work-home), and so on. It is important to note that although one covers the work-home path almost each day, we still have only one example, since these repeated examples cannot be considered independent samples from the underlying distribution of all the work-home paths. In addition, many user related events, for example, car accidents, are also very rare relative to drift speed, which can be rather fast due to, for example, weather conditions. With our method – in a very large network – one can still produce an up-to-date accident prediction

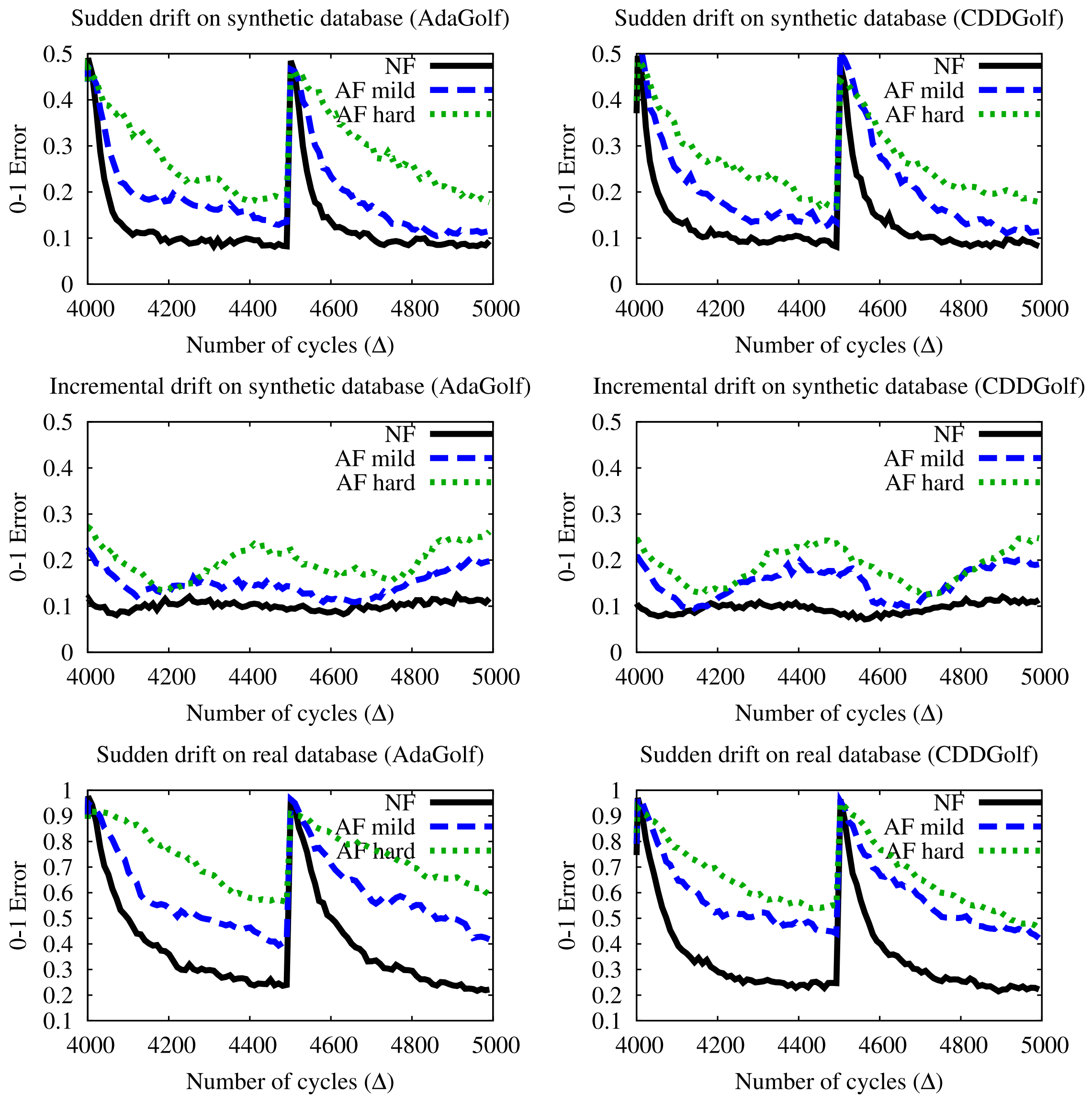

Figure 5.6. Prediction performance under failure.

model based on these examples. Finally, in cases when users need to enter data explicitly, it is very likely that the average rate of samples will be relatively very low.

5.5.4 Fault Tolerance

We performed simulations with the failure scenarios described previously. Figure 5.6 contains the results. From this we can observe that the effect of the fail-

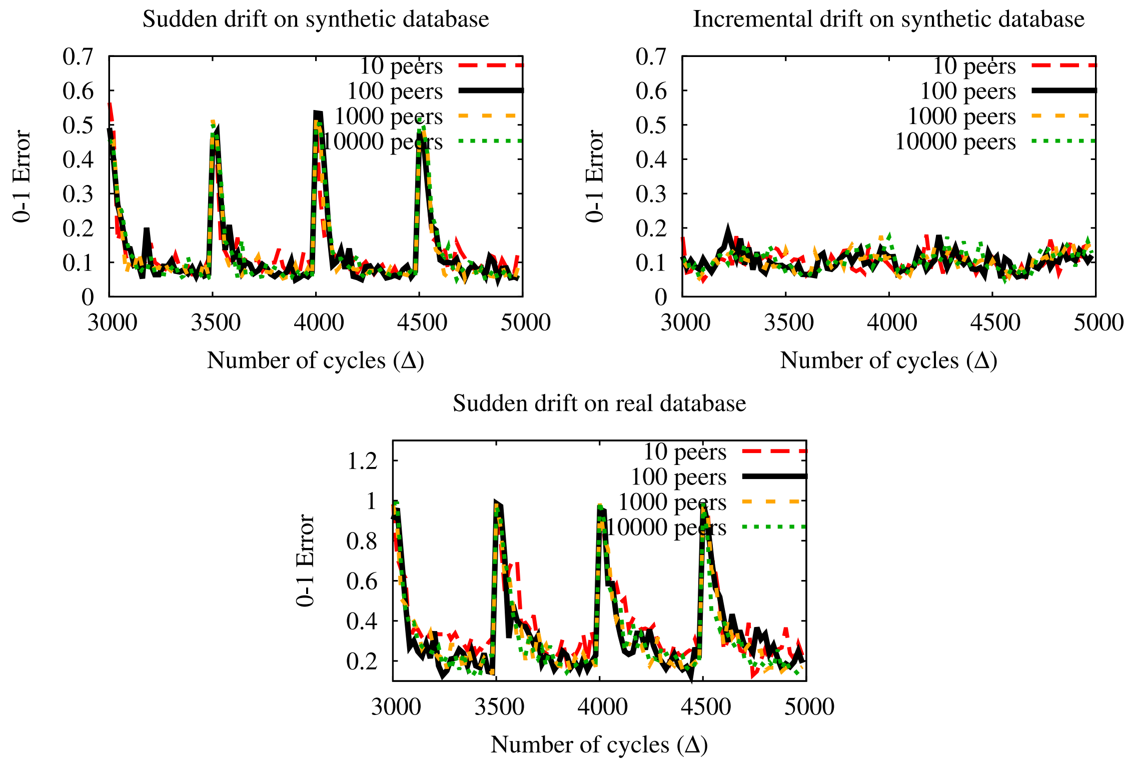

Figure 5.7. The effect of network size (with sampling rate $1/\Delta$).

ures is a slower convergence speed. This effect can mostly be accounted for by the message delay, since all the random walks will be proportionally slower. This has the same effect as if the cycle length Δ was proportionally larger in a failure-free scenario.

5.5.5 Scalability

In Figure 5.7 we present the results of ADAGOLF in different network sizes. For CDDGOLF we obtain identical results (not shown). We cannot identify any significant effect of the network size in most of the scenarios. In the case of the real dataset (that is harder to learn) we can realize that larger networks result in a slightly better performance, which is most likely due to the fact that more independent samples are available in larger networks.

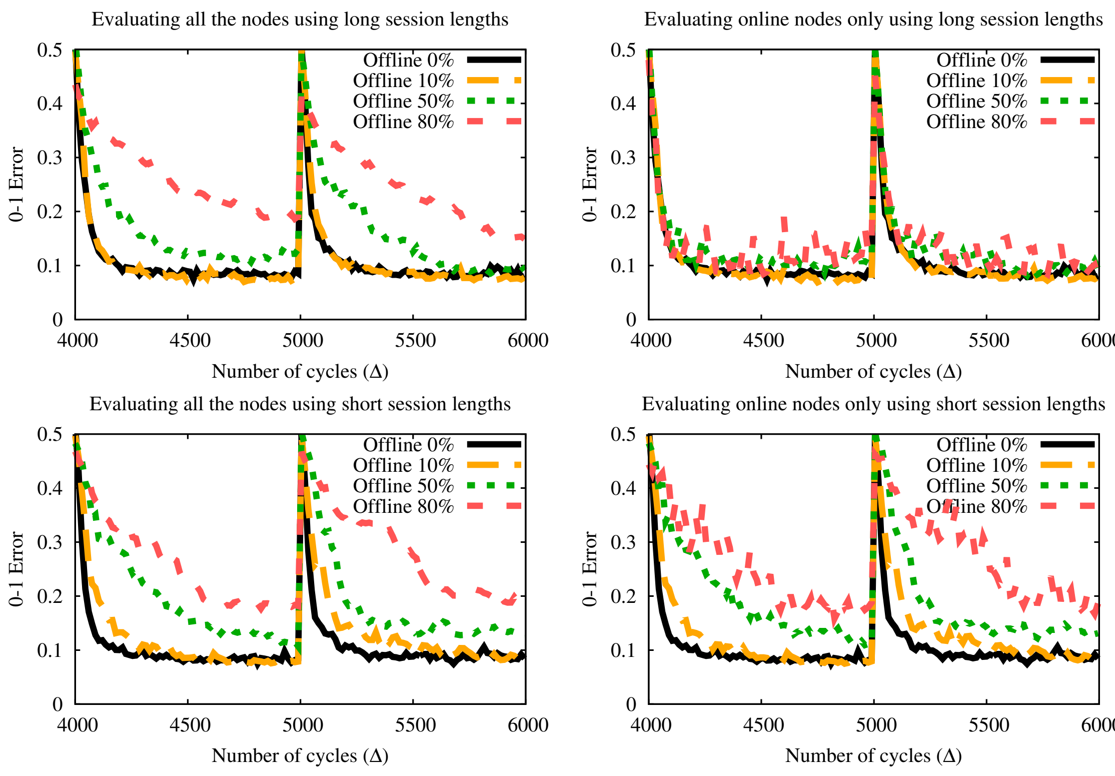

Figure 5.8. The effect of the churn.

5.5.6 Churn Revisited

Our last set of experiments focuses on the effect of churn on AdaGoLF. Our approach to modeling churn is described in Section 5.4.5. We briefly mentioned there that we opted for modeling extremely short online session lengths as a worst case scenario. Here we justify this decision by showing that indeed if session lengths are longer (closer to the realistic value), then the performance is very close to that of the scenario without churn. For this reason, we added a scenario where the average session length was 100Δ. Note that this still results in $\Delta = 3$ minutes, which is still very long.

Figure 5.8 shows the results. In this figure we show our experiments with 0%, 10%, 50% and 80% of the peers offline on average at any given time. In addition, we also present the 0-1 error over both the set of all the nodes (including those that are offline) and over the set of online nodes.

It is clear from the plot that increasing the proportion of the nodes that are offline results in a slower convergence if we consider also the nodes that are offline. In the case of long session lengths, this is because nodes that are offline are unable to adapt to concept drift, and hence will have a very poor prediction performance after their models get out of date. However, if we consider only the performance of the online nodes, then the performance remains essentially the same even for the highest proportion of offline nodes. This is because nodes are online long enough (compared to the speed of convergence) so that most models in the system are able to perform long random walks without getting stuck on a node that is going offline before reaching convergence.

In the case of extremely short short session lengths, it does not matter whether we examine online or offline nodes. In scenarios with a high proportion of offline nodes convergence will slow down anyway, because most random walks will be able to make only one or two steps before getting stuck on a node that is going offline, which results in an overall slowdown of the convergence of all the models. Note however, that this scenario is highly unrealistic.

Another interesting aspect of churn is its effect on the age distribution of models. The case without churn is discussed theoretically in Section 6.4. Figure 5.9 shows the histogram of the model age distribution in the case of churn, averaged over a set of 10,000 snapshots during the simulation, which were taken in every 10th cycle during a simulation of 100,000 cycles. Note that model age is defined by the number of samples a given model has been trained on (as opposed to the number of cycles elapsed after its initialization). We can observe that the age distribution remains very similar to the case without churn. As before, this is due to the fact that churn events are rare so they do not interfere with the ageing of models much. However, in Figure 5.10 we can see that if session lengths are extremely short, then the distribution gets biased towards the younger models. Note that the histograms that consider all the nodes (including those that are offline) are very similar (not shown).

5.6 Conclusions

In this paper we proposed adaptive versions of our GoLF framework: AdaGoLF and CDDGoLF. In the case of AdaGoLF the adaptivity is implemented in a very

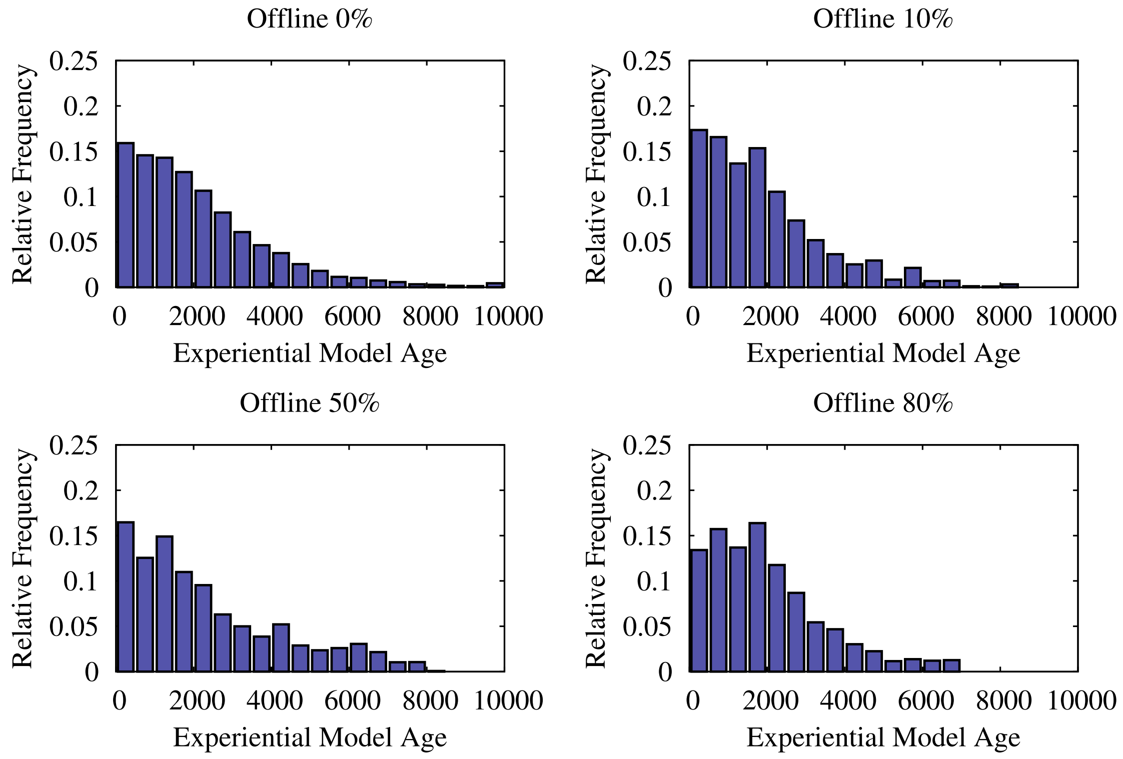

Figure 5.9. Model age histograms over online nodes for long session lengths.

simple manner through the management of the age distribution of the models in the network, making sure that there is a sufficient diversity of different ages in the pool. This is not a usual approach, as most of the related work focus on building and combining local models, and on detecting and predicting drift explicitly. CDDGoLF also restarts some of the models, but this decision is not blind as in the case of AdaGoLF, but instead it is based on the performance history of the model in question.

We performed a thorough experimental study in which we compared AdaGoLF and CDDGoLF with a set of baseline algorithms that represented idealized versions of the main techniques that are applied in related work. Our main conclusion is that in those scenarios, where the sampling rate from the underlying distribution is low relative to the speed of drift, our solutions clearly outperform all the baseline solutions, approximating the "God's Eye view" model, which represents the best possible performance.

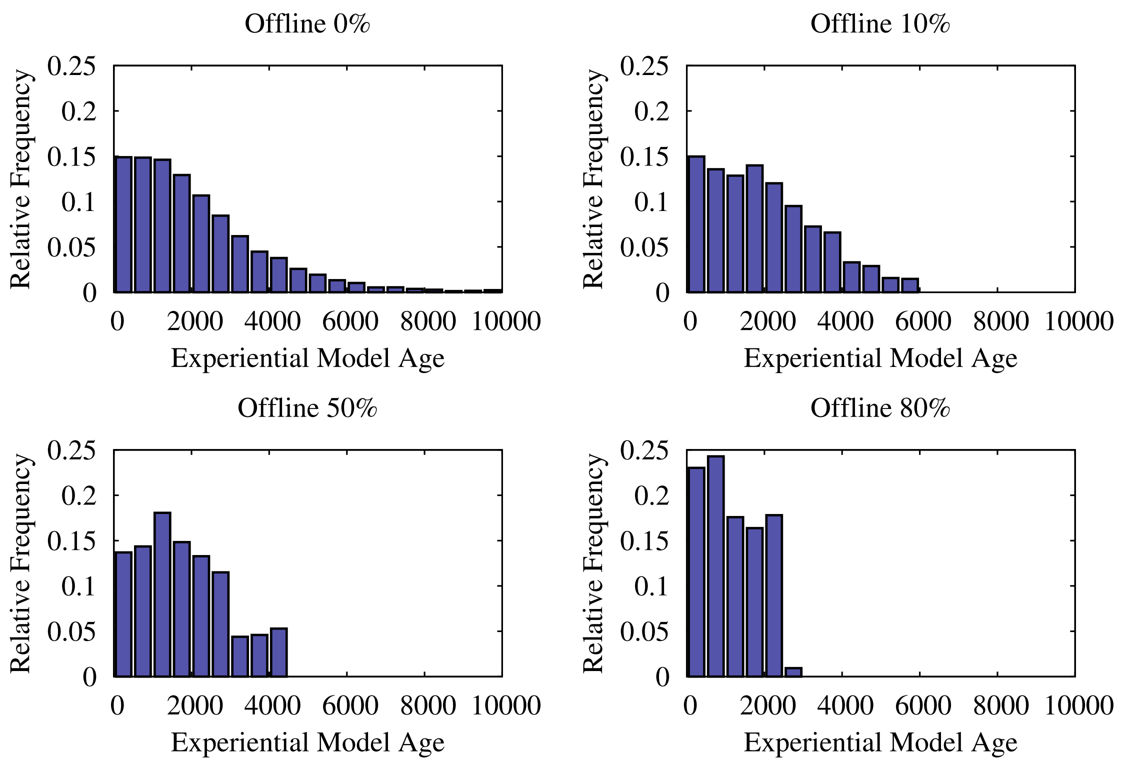

Figure 5.10. Model age histograms over online nodes for short session lengths.

One surprising observation is that, although completely blind, ADAGOLF has practically the same performance as CDDGOLF in the scenarios and databases we examined. This suggests that it is an attractive option if we need to issue no explicit alarms in the case of drift. CDDGOLF detects drift explicitly, but its implementation is slightly more complicated, and it could in principle be sensitive to certain patterns of drift since its drift detection heuristic depends on the drift pattern.

We also indicated that our algorithms can be enhanced to deal with (or rather, be robust to) higher sample rates as well, although in that case purely local model building can also be sufficient.

CHAPTER 6

Singular Value Decomposition

Finding a low-rank decomposition of a matrix is an essential tool in data mining and information retrieval [10]. Prominent applications include recommender systems [34], information retrieval via Latent Semantic Indexing [17, 91], Kleinberg's HITS algorithm for graph centrality [65], clustering [32, 80], and learning mixtures of distributions [3, 61].

Collaborative filtering forms one of the most prominent applications of low rank matrix approximation. To implement a recommender algorithm, one can define a matrix A of dimensions $m \times n$ with one row assigned to one of m users, and one column assigned to one movie out of a set of n movies. The essence of the ratings matrix A is given by a low rank decomposition $A \approx XY^T$, where matrices X and Y^T are of dimensions $m \times k$ and $k \times n$, respectively, and where k is small [68]. A row of X can be interpreted as a compressed representation (features) of the corresponding user, and a column of Y^T contains features of a movie. This representation can be applied to calculate missing ratings to offer recommendations. In addition to recommender systems, low rank approximation finds applications in summarizing adjacency matrices for social network analysis or term-document matrices for text classification.

Data such as movie ratings, lists of friends, or personal documents is often very sensitive. It is argued in [85] that it is paramount for privacy that users keep their data (e.g. ratings and user factors) local. Instead of uploading such data to cloud servers, it is a natural idea to store it only on personal devices and process it in place in a fully distributed way suitable for P2P services. This would increase the level of privacy and remove any dependence on central infrastructure and services. However, such an algorithm has to be extremely robust as networks of personal devices are unreliable with users entering and leaving dynamically. In addition, the communication costs should remain affordable as well.

The problem we tackle is implementing singular value decomposition (SVD) – a popular method for low rank approximation – in large fully distributed systems in a robust and scalable manner. We assume that the matrix to be approximated is stored in a large network where each node knows one row of the matrix (personal attributes, documents, media ratings, etc). We do not allow this personal information to leave the node, yet we want the nodes to collaboratively compute the SVD. Methods applied in large scale distributed systems such as synchronized parallel gradient search or distributed iterative methods are not preferable in our system model due to their requirements of synchronized rounds or their inherent issues with load balancing.

6.1 Contributions

To meet these goals, we implement singular value decomposition (SVD), an approach to low rank decomposition with the attractive property that the matrices X and Y in the decomposition consist of orthogonal vectors. We present a stochastic gradient descent (SGD) algorithm to find the SVD, where several instances of the matrix Y perform a random walk in the network visiting the data (the rows of A), while matrices A and X are stored in a fully distributed way with each row at different nodes. The rows of matrices A and X are accessed only locally by the node that stores them. Note that the public matrix Y carries no sensitive information. When Y visits a node, it gets updated based on the local row of A, and the local row of X gets updated as well.

To the best of our knowledge, we present the first fully distributed SGD algorithm that keeps X and A strictly local. As a special feature, our algorithm updates several versions of Y by sending them around over the network, resulting in a convergence much faster than a single instance of SGD for SVD. The algorithm is fully asynchronous: messages can be delayed or lost. We only rely on random walks that can perform every transition in a bounded time. We show that the only stable fix points of the search algorithm correspond to the SVD. In addition, we perform an experimental analysis and demonstrate the convergence speed and the scalability of the protocol.

6.2 Related Work

Calculating the SVD is a well studied problem. One approach is based on considering it as an optimization problem (see Section 6.3) and using *gradient search* to solve it [43]. In general, parallel versions of gradient search often assume the MapReduce framework [23] or a similar, less constrained, but still centralized model [72]. In these approaches, partial gradients are calculated over batches of data in parallel, and these are either applied to blocks of X and Y in the map phase or summed up in the reduce phase. Zinkevich et al. propose a different approach in which SGD is applied on batches of data and the resulting models are then combined [115]. Gemulla et al. [41] propose an efficient parallel technique in which blocks of X and Y are iteratively updated while only blocks of Y are exchanged. In contrast to these approaches, we work with fully distributed data: we do not wish to make X public, and we do not rely on central components or synchronization, a set of requirements ruling out the direct application of earlier approaches.

Another possibility is using fully distributed *iterative methods*. GraphLab [75] supports iterative methods for various problems including SVD. In these approaches, the communication graph in the network is defined by the non-zero elements of matrix A, in other words, A is stored as edge-weights in the network. This is feasible only if A is (very) sparse and well balanced; a constraint rarely met in practice. In addition, iterative methods need access to A^T as well, which violates our constraint that the rows of A are not shared. Using the same edge-weight representation of A, one can implement another optimization approach for matrix decomposition: an iterative optimization of subproblems over overlapping local subnetworks [73]. The drawback of this approach is, again, that the structure of A defines the communication network and access to A^T is required. The approach of Ling et al. [74] also needs global information: in each iteration step a global average needs to be calculated.

The first fully distributed algorithm for spectral analysis was given in [64] where data is kept at the compute nodes and partial results are sent through the network. That algorithm, however, only computes the user side singular vectors but not the item side ones and hence insufficient, for example, to provide recommendations. Similarly, [67] computes the low rank approximation but not the decomposition. This drawback is circumvented in [56] by assigning compute nodes not just for users but for items as well. In their gradient descent algorithm, item vectors are also stored at peers, which means that all items must know the ratings they receive, and this violates privacy.

We overcome the limitations of earlier fully distributed and gossip SVD algorithms by providing an algorithm without item-based processing elements, i.e. our algorithm is fully distributed in the sense that processing is exclusively done at the users, who may access their own ratings and approximations of the item factor vectors.

6.3 Problem Definition

6.3.1 Low-Rank and Singular Value Decomposition

The problem we tackle is rank-k matrix approximation. We are given a matrix $A \in \mathbb{R}^{m \times n}$. Matrix A holds our raw data, such as ratings of movies by users, or word statistics in documents. We are looking for matrices $X \in \mathbb{R}^{m \times k}$ and $Y \in \mathbb{R}^{n \times k}$ such that the error function

$$J(X,Y) = \frac{1}{2}\|A - XY^T\|_F^2 = \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n}(a_{ij} - \sum_{l=1}^{k} x_{il}y_{jl})^2 \tag{6.1}$$

is minimized. We say that the matrix XY^T for which this error function is minimized is an optimal rank-k approximation of A. Clearly, matrices X and Y^T – and hence XY^T – have a rank of at most k. Normally we select k such that $k \ll \min(m, n)$ in order to achieve a significant compression of the data. As a result, matrices X and Y^T can be thought of as high level features (such as topics of documents, or semantic categories for words) that can be used to represent the original raw data in a compact way.

Singular value decomposition (SVD) is related to the above matrix decomposition problem. The SVD of a matrix $A \in \mathbb{R}^{m\times n}$ involves two orthogonal matrices $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ such that

$$A = U\Sigma V^T = \sum_{i=1}^{r} \sigma_i u_i v_i^T = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \cdots + \sigma_r u_r v_r^T \tag{6.2}$$

where the columns of the matrices $U = [u_1 u_2 \cdots u_m]$ and $V = [v_1 v_2 \cdots v_n]$ are the left and right singular vectors, and $\Sigma \in \mathbb{R}^{m\times n}$ is a diagonal matrix containing the singular values $\sigma_1, \sigma_2, \ldots, \sigma_r \geq 0$ of A ($r = \min(m, n)$). The relationship of SVD and low rank decomposition is that $U_k \Sigma_k V_k^T$ is an optimal rank-k approximation of A, where the matrices $U_k \in \mathbb{R}^{m\times k}$ and $V_k \in \mathbb{R}^{n\times k}$ are derived from U and V by keeping the first k columns, and $\Sigma_k \in \mathbb{R}^{k\times k}$ is derived from Σ by keeping the top left $k \times k$ rectangular area, assuming without loss of generality that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r$ [102].

Our goal in this paper is to find $X^* = U_k \Sigma_U$ and $Y^* = V_k \Sigma_V$ such that Σ_U and Σ_V are diagonal and $\Sigma_U \Sigma_V = \Sigma_k$. In other words, although X^* and Y^* are not uniquely defined, we require them to contain orthogonal columns that are scaled versions of left and right singular vectors of A, respectively.

6.3.2 Data Distribution

As of data distribution, we assume that each node has exactly one row (but note that our algorithms can be applied – and in fact profit from it – if a node has several rows). Our data distribution model is motivated by application scenarios in which potentially sensitive data is available locally, such as private documents or ratings that naturally define rows of a matrix A, but where a decomposition

Algorithm 6.15 P2P low-rank factorization at node i

```
1: $a_i$                                        ▷ row $i$ of $A$
2: initialize $Y$
3: initialize $x_i$                             ▷ row $i$ of $X$
4: loop
5:     wait($\Delta$)
6:     $p \leftarrow$ selectPeer()
7:     send $Y$ to $p$
8: end loop
9: procedure ONRECEIVEY($\tilde{Y}$)
10:    $Y \leftarrow \tilde{Y}$
11:    $(Y, x_i) \leftarrow$ update($Y, x_i, a_i$)
12: end procedure
```

needs to be found collectively without blatantly exposing this private data.

6.4 Algorithm

Our algorithm has its roots in the GOLF framework [89]. Algorithm 6.15 contains a version of the GOLF algorithm adopted to our problem. Each node i has its own approximation of the full matrix Y^* and an approximation of row i of X^*: x_i. Thus, the nodes collectively store one version of the matrix X (the approximation of X^*) distributed just like matrix A with each node storing one row. At the same time, at every point in time every node has a possibly different approximation of the full matrix Y^* locally.

These different approximations of Y^* perform random walks in the network and get updated at the visited nodes using the local data of the given node (a row of A and X). First, each node i in the network initializes Y and x_i uniformly at random from the interval $[0, 1]$. The nodes then periodically send their current approximation Y to a randomly selected peer from the network. The period is denoted by Δ. To select a random peer, we rely on a peer sampling service as mentioned in Section 2.3. When receiving an approximation $\tilde{Y}$ (see procedure ONRECEIVEY) the node stores this approximation and subsequently it updates both Y and x_i using a stochastic gradient rule.

Algorithm 6.16 rank-k update at node i

```
1: $\eta$                                        ▷ learning rate
2: procedure UPDATE($Y, x_i, a_i$)
3:     err $\leftarrow a_i - x_i Y^T$
4:     $x_i' \leftarrow x_i + \eta \cdot \mathrm{err} \cdot Y$
5:     $Y' \leftarrow Y + \eta \cdot \mathrm{err}^T \cdot x_i$
6:     return $(Y', x_i')$
7: end procedure
```

The algorithm involves periodic message sending at every node with a period of Δ. Later on, when we refer to one *iteration* or *round* of the algorithm, we simply mean a time interval of length Δ. Note that we do not require any synchronization of rounds over the network. Messages are sent independently, and they can be delayed or dropped as well. We require only that the delays are bounded and that the message drop probability is less than one. This allows the random walks to progress.

Algorithm 6.15 requires an implementation of the procedure UPDATE. We will now elaborate on two different versions that implement different stochastic gradient update rules.

6.4.1 Update Rule for General Rank-k Factorization

Let us first consider the error function given in equation (6.1) and derive an update rule to optimize this error function. The partial derivatives of $J(X, Y)$ by X and Y are

$$\frac{\partial J}{\partial X} = (XY^T - A)Y, \quad \frac{\partial J}{\partial Y} = (YX^T - A^T)X. \tag{6.3}$$

Since only x_i is available at node i, the gradient is calculated only w.r.t. x_i instead of X. Accordingly, the stochastic gradient update rule with a learning rate η can be derived by substituting x_i as shown in Algorithm 6.16. Although function J is not convex, it has been shown that all the local optima of J are also global [102]. Thus, for a small enough η, any stable fix point of the dynamical system implemented by Algorithm 6.15 with the update rule in Algorithm 6.16 is guaranteed to be a global optimum.

6.4.2 Update Rule for Rank-k SVD

Apart from minimizing the error function given in Equation (6.1) let us now also set the additional goal that the algorithm converges to X^* and Y^*, as defined in Section 6.3. This is indeed a harder problem: while (X^*, Y^*) minimizes (6.1), any pair of matrices $(X^* R^{-1}, Y^* R^T)$ will also minimize it for any invertible matrix $R \in \mathbb{R}^{k\times k}$, so Algorithm 6.16 will not be sufficient.

From now on, we will assume that the non-zero singular values of A are all unique, and that the rank of A is at least k. That is, $\sigma_1 > \cdots > \sigma_k > 0$. This makes the discussion simpler, but these assumptions can be relaxed, and the algorithm is applicable even if these assumptions do not hold.

Our key observation is that any optimal rank-1 approximation $X_1 Y_1^T$ of A is such that $X_1 \in \mathbb{R}^{m\times 1}$ contains the (unnormalized) left singular vector of A that belongs to σ_1, the largest singular value of A. Similarly, Y_1 contains the corresponding right singular vector. This is because for any optimal rank-k approximation XY^T there is an invertible matrix $R \in \mathbb{R}^{k\times k}$ such that $X = X^* R$ and $Y^T = R^{-1} {Y^*}^T$[102]. For $k = 1$ this proves our observation because, as defined in Section 6.3.1, $X^* \sim u_1$ and $Y^* \sim v_1$. Furthermore, for $k = 1$,

$$X_1 Y_1^T = X^* {Y^*}^T = \sigma_1 u_1 v_1^T, \tag{6.4}$$

which means that (using Equation (6.2)) we have

$$A - X_1 Y_1^T = \sum_{i=2}^{r} \sigma_i u_i v_i^T. \tag{6.5}$$

Thus, a rank-1 approximation of the matrix $A - X_1 Y_1^T$ will reveal the direction of the singular vectors corresponding to the second largest singular value σ_2. A naive approach based on these observations would be to first compute $X_1 Y_1^T$, a rank-1 approximation of A. After this has converged, we could compute a rank-1 approximation $X_2 Y_2^T$ of $A - X_1 Y_1^T$. This would give us a rank-2 approximation of A containing the first two left and right singular vectors, since according to the above observations $[X_1, X_2][Y_1, Y_2]^T$ is in fact such a rank-2 approximation. We could repeat this procedure k times to get the desired decomposition X^* and Y^*

Algorithm 6.17 rank-k SVD update at node i

```
1: $\eta$                                          ▷ learning rate
2: procedure UPDATE($Y, x_i, a_i$)
3:     $a'_i \leftarrow a_i$
4:     for $\ell = 1$ to $k$ do                    ▷ $y_\ell$ : column $\ell$ of $Y$
5:         err $\leftarrow a'_i - x_{i\ell} \cdot y_\ell^T$
6:         $x'_{i\ell} \leftarrow x_{i\ell} + \eta \cdot$ err $\cdot y_\ell$
7:         $y'_\ell \leftarrow y_\ell + \eta \cdot \mathrm{err}^T \cdot x_{i\ell}$
8:         $a'_i = a'_i - x_{i\ell} \cdot y_\ell^T$
9:     end for
10:    return $(Y', x'_i)$
11: end procedure
```

by filling in one column at a time sequentially in both matrices.

A more efficient and robust approach is to let all rank-1 approximations in this sequential naive approach evolve at the same time. Intuitively, when there is a reasonable estimate of the singular vector corresponding to the largest singular value, then the next vector can already start progressing in the right direction, and so on. This idea is implemented in Algorithm 6.17.

6.4.3 Synchronized Rank-k SVD

As a baseline method in our experimental evaluation, we will use a synchronized version of Algorithm 6.15 with the update rule in Algorithm 6.17. In this version (shown in Algorithm 6.18), the rank-1 updates are done over the entire matrix A at once, and there is only one central version of the approximation of Y as opposed to the several independent versions we had previously. As in Algorithm 6.15, the matrices X and Y are initialized with uniform random values from $[0, 1]$. Note that this algorithm listing uses a different notation: here x_ℓ denotes the ℓ-th column, not the ℓ-th row.

Note that – although it is formulated as a centralized sequential algorithm – this algorithm can easily be adapted for the MapReduce framework, where the mappers compute parts of the gradients (e.g., the gradients of the rows of A as in Algorithm 6.17) while the reducer sums up the components of the gradient and executes the update steps.

Algorithm 6.18 Iterative synchronized rank-k SVD

```
1: $A$                                          ▷ The matrix to be factorized
2: $\eta$                                        ▷ learning rate
3: initialize $Y$
4: initialize $X$
5: while not converged do
6:     $A' = A$
7:     for $\ell = 1$ to $k$ do
8:         err $\leftarrow A' - x_\ell \cdot y_\ell^T$        ▷ $x_\ell$ : column $\ell$ of $X$
9:                                              ▷ $y_\ell$ : column $\ell$ of $Y$
10:        $x'_\ell \leftarrow x_\ell + \eta \cdot \mathrm{err} \cdot y_\ell$
11:        $y'_\ell \leftarrow y_\ell + \eta \cdot \mathrm{err}^T \cdot x_\ell$
12:        $A' = A' - x_\ell \cdot y_\ell^T$
13:    end for
14:    $X = X'$;  $Y = Y'$
15: end while
```

6.5 Experiments

Here we demonstrate various properties of our algorithm including convergence speed, scalability and robustness. Our testbed of matrices includes standard machine learning test data sets as well as synthetic matrices with controllable properties.

In the case of the distributed algorithms the number of nodes in the network equals the number of rows of the matrix to be factorized, since every node has exactly one row of the matrix. We used the PeerSim [84] simulator with the event-based engine, and we implemented the peer sampling service by the NEWSCAST [107] protocol.

6.5.1 Algorithms

Here we provide names for the algorithms we include in our experiments. Algorithm 6.15 with the update rule in Algorithm 6.17 (our main contribution) will be referred to as Fully Distributed SVD (FUDISVD). Replacing Algorithm 6.17 with Algorithm 6.16 we get Fully Distributed Low Rank Decomposition (FUDILRD). Recall that this algorithm will converge to a rank-k decomposition that is not necessarily the SVD of A. Algorithm 6.18 will be called Gradient SVD (GRADSVD). Recall that this algorithm can be parallelized: for example, the gradient can be calculated row by row and then summed up to get the full gradient.

Finally, we introduce a baseline algorithm, Stochastic Gradient SVD (SGSVD). This algorithm uses the update rule in Algorithm 6.17 but we have only a single approximation Y at any point in time, and there is only one process which repeatedly gets random rows of A and then applies the update rule in Algorithm 6.17 to the current approximation Y and the corresponding row of X.

6.5.2 Error Measures

Cosine similarity To measure the difference between the correct and the approximated singular vectors, we used cosine similarity, because it is not sensitive to the scaling of the vectors (recall that our algorithm does not guarantee unit-length columns in X and Y). The formula to measure the error of a rank-k decomposition XY^T is

$$\mathrm{Error}(X, Y) = \frac{1}{2k} \sum_{i=1}^{k} 1 - \left| \frac{y_i^T v_i}{\|y_i\|} \right| + 1 - \left| \frac{x_i^T u_i}{\|x_i\|} \right|, \tag{6.6}$$

where x_i, y_i, u_i and v_i are column vectors of X, Y, U_k and V_k, respectively. Matrices U_k and V_k are orthogonal matrices defined in Section 6.3.

Frobenius norm Another measure of error is given by function $J(X, Y)$ defined in Equation (6.1). The advantage of this measure is that it can be applied to Algorithm 6.16 as well. However, obviously, this error measure does not reflect whether the calculated matrices X and Y contain scaled singular vectors or not; it simply measures the quality of rank-k approximation. On the plots we call this error measure FNORM.

6.5.3 Data Sets

Synthetic data We included experiments on synthetic data so that we can evaluate the scalability and the fault tolerance of our method in a fully controlled way. We first generated random singular vectors producing matrices U and V with the

Table 6.1. The main properties of the real data sets

	Iris	Pendigits	Segmentation
Number of instances (m)	150	10992	2310
Number of features (n)	4	16	19
Minimal k such that $\sum_{i=1}^{k} \sigma_i^2 \,/\, \sum_{i=1}^{n} \sigma_i^2 > 0.9$	2	10	5

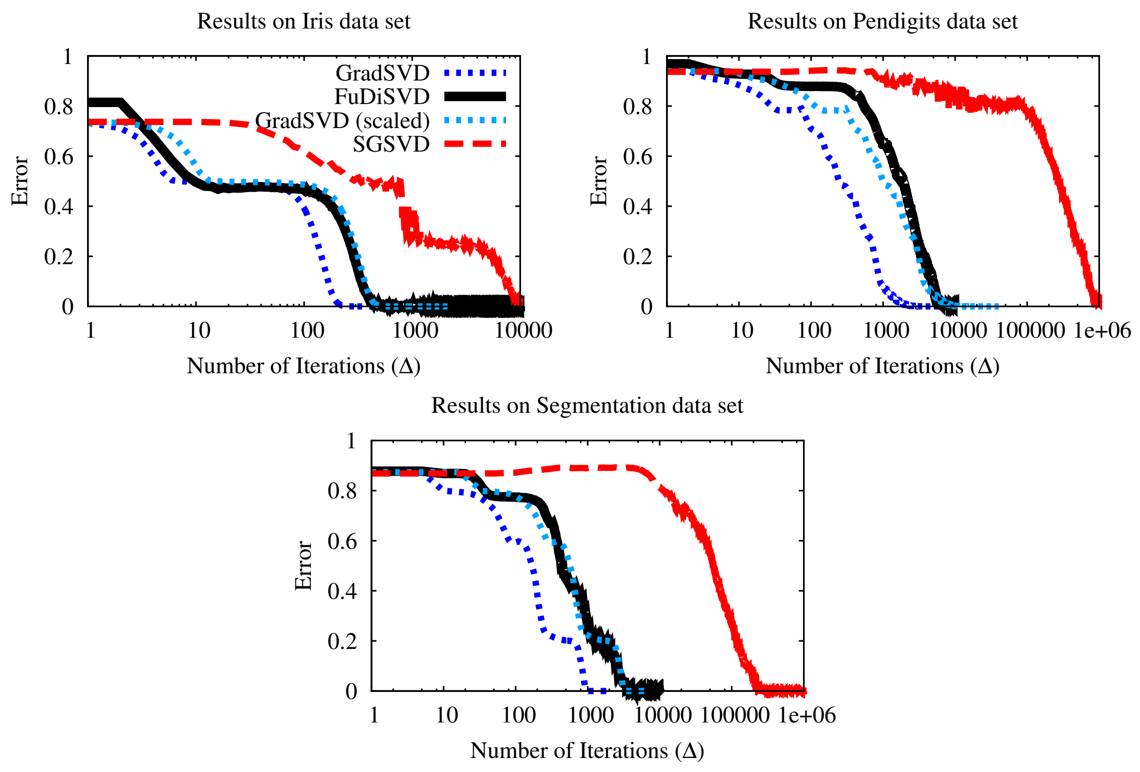

Figure 6.1. Convergence on the real data sets. Error is based on cosine similarity. In the scaled version of GRADSVD the number of iterations is multiplied by $\log_{10} m$ (see text).

help of the butterfly technique [42]. Since matrices to be decomposed often originate from graphs, and since the node degrees and the spectrum of real graphs usually follow a power law distribution [24, 81], the expected singular values in the diagonal of Σ were generated from a Pareto distribution with parameters $x_m = 1$ and $\alpha = 1$. This way we construct a matrix $A = U\Sigma V^T$ where we know and control the singular vectors and values.

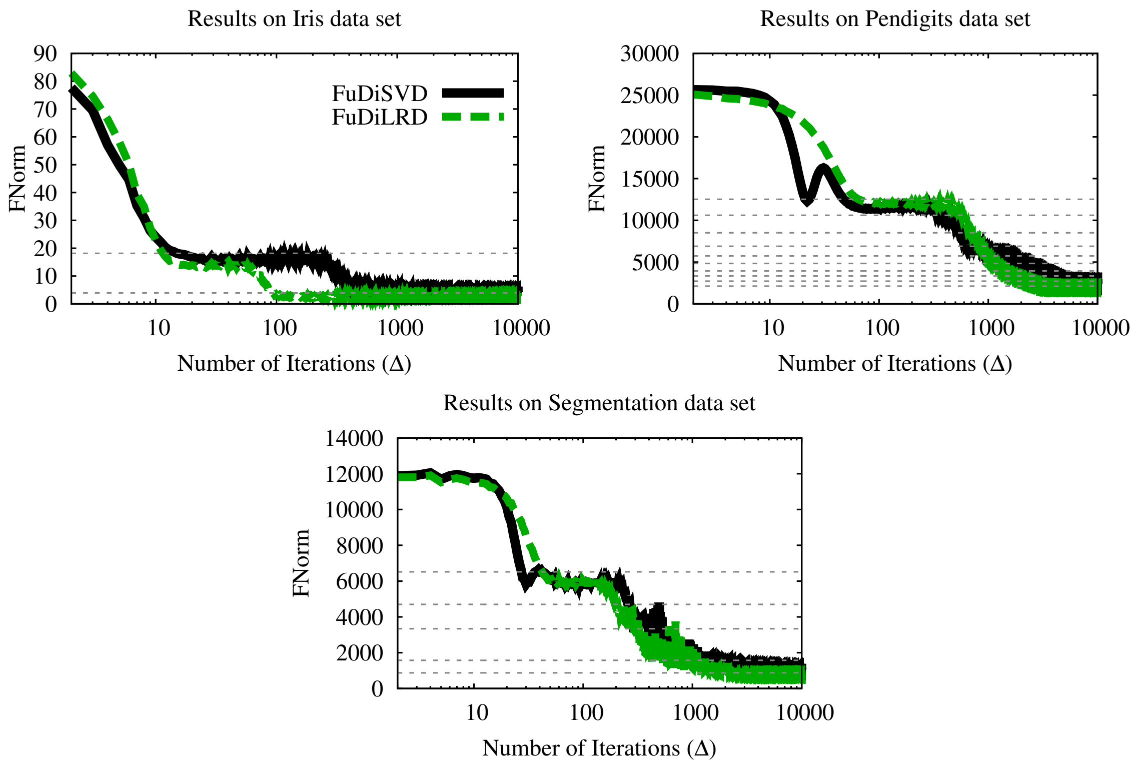

Figure 6.2. Convergence on the real data sets. Error is based on the Frobenius norm. Horizontal dashed lines in top-down order show the FNORM value for the optimal rank-i approximations for $i = 1, \ldots, k$.

Real data These matrices were constructed from data sets from the well known UCI [12] machine learning repository. In these applications the role of SVD is dimensionality reduction and feature extraction. The selected data sets are the Iris, the Pendigits (Pen-Based Recognition of Handwritten Digits) and the Segmentation (Statlog (Image Segmentation)) data sets. Parameter k was set so that the approximation has at least 90% of the information in the data set, that is, we set the minimal k such that $\sum_{i=1}^{k} \sigma_i^2 / \sum_{i=1}^{n} \sigma_i^2 > 0.9$ [4]. Table 6.1 illustrates the main properties of the selected data sets. In order to be able to compute the error over these matrices, we computed the exact singular value decomposition using the Jama [86] library.

6.5.4 Convergence

The experimental results are shown in Figures 6.1 and 6.2. We tested the convergence speed of the algorithms over the real data sets with parameter k set according to Table 6.1.

Figure 6.1 illustrates the deviation from the exact singular vectors. GRADSVD is the fastest to converge, however, it either needs a central database to store A, or it requires a synchronized master-slave communication model when parallelized. We also show a reference curve that is calculated by scaling the number of iterations by $\log_{10} m$ for GRADSVD. The motivation is a more scalable potential implementation of GRADSVD in which there is a hierarchical communication structure in place where nodes propagate partial sums of the gradient up a tree (with a branching factor of 10) instead of each node communicating with a central server as in [16]. This curve almost completely overlaps with that of FUDISVD, our fully distributed robust algorithm.

We also illustrate the speedup w.r.t. SGSVD. The major difference between SGSVD and FUDISVD is that in FUDISVD there are m different approximations of Y all of which keep updating the rows of X simultaneously, while in SGSVD there is only one version of Y. Other than that, both algorithms apply exactly the same gradient update rule. In other words, in FUDISVD any row of X experiences m times as many updates in one unit of time.

Figure 6.2 illustrates the difference between FUDILRD and FUDISVD. Both optimize the same error function (that is shown on the plots), however, FUDISVD introduces the extra constraint of aiming for the singular vectors of A. Fortunately this extra constraint does not slow down the convergence significantly in the case of the data sets we examined, although it does introduce a bumpier convergence pattern. The reason is that the singular vectors converge in a sequential order, and the vectors that belong to smaller singular values might have to be completely re-oriented when the singular vectors that preceed them in the order of convergence have converged.

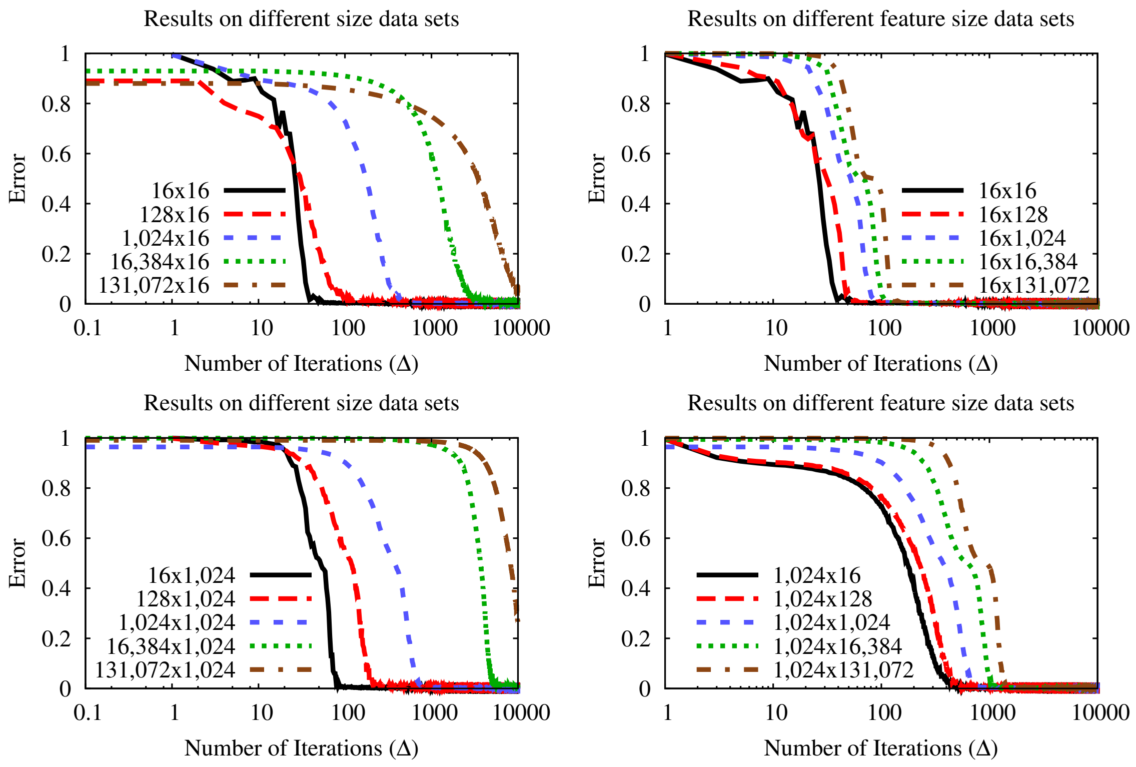

Figure 6.3. Results on synthetic data sets using networks of different dimensions. We set $k = 1$, and all the matrices had a rank of 16.

6.5.5 Scalability

For evaluating the scalability of FUDISVD we generated a range of synthetic matrices of various sizes using the method described earlier. Figure 6.3 shows the outcome of the experiments. Clearly, the method is somewhat more sensitive to changing the number of nodes (that is, to varying the first dimension m) than to varying the second dimension n (the length of a row). This is not surprising as the full row of A is always used at once in an update step, irrespective of its length, whereas a larger m requires visiting more nodes, which takes more iterations.

However, when m is large, we can apply *sampling techniques* [33]. That is, we can consider only a small sample of the network drawn uniformly or from an appropriately biased distribution and calculate a high quality SVD based on that subset. To illustrate such sampling techniques, we implemented uniform sampling. When we wish to select a given proportion p of the nodes, each node

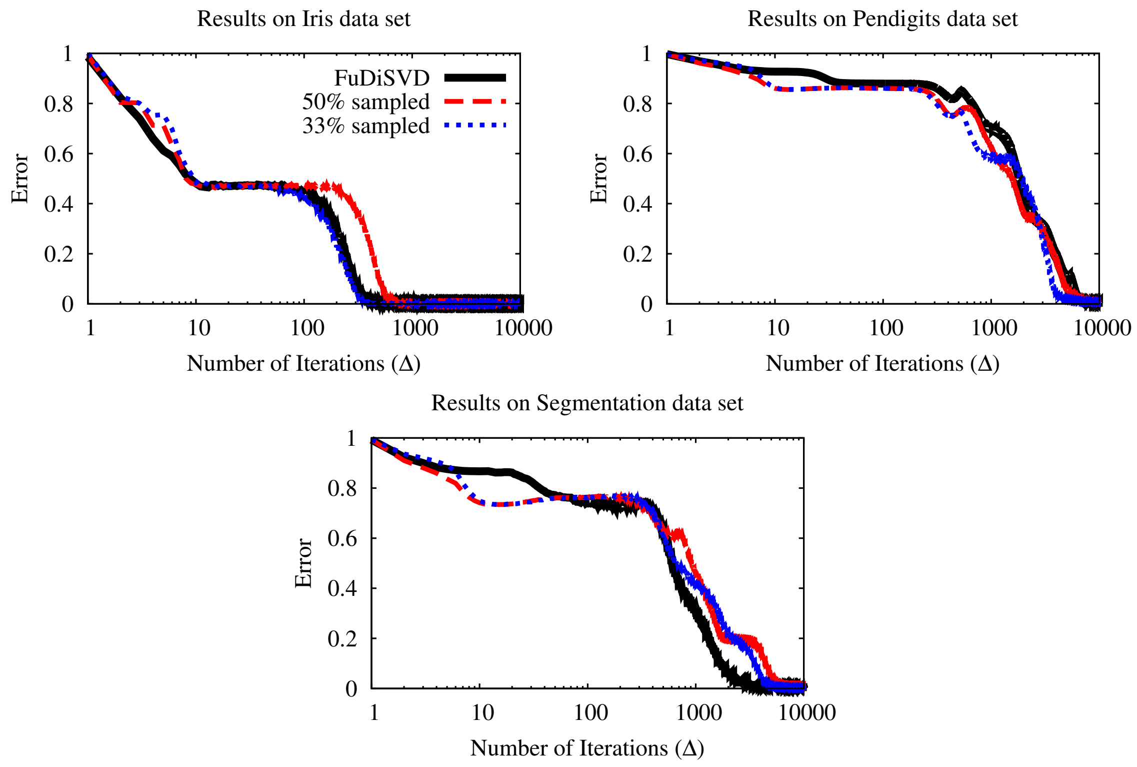

Figure 6.4. Results when only the 50/33% randomly sampled instances were used from the data set.

decides locally about joining the sample with a probability p. The size of the resulting sample will follow the binomial distribution $B(N, p)$.

Figure 6.4 shows our experimental results with $p = 1/2$ and $p = 1/3$. Clearly, while communication costs overall are decreased proportionally to the sample size, on our benchmark both precision and convergence speed remain almost unchanged. The only exception is the Iris data set. However, that is a relatively small data set over which the variance of our uniform sampling method is relatively high (note, for example, that the run with $p = 1/3$ resulted in a better performance than with $p = 1/2$).

6.5.6 Failure Scenarios

We used two different failure scenarios: a mild and a hard one. In the two scenarios message delay was drawn uniformly at random from between Δ and 5Δ or

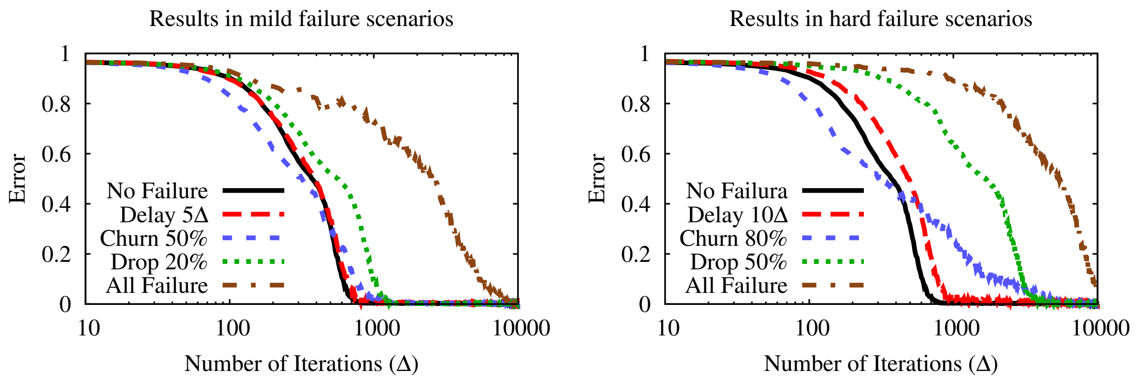

Figure 6.5. Results in different failure scenarios using a 1024×1024 synthetic matrix with a rank of 16. We set $k = 1$.

10Δ time units, respectively. Messages were dropped with a probability of 20% or 50%, respectively.

Node churn was modeled taking into account statistics from a BitTorrent trace [97] as well as known empirical findings [105]. We draw the online session length for each node independently from a lognormal probability distribution with parameters $\mu = 5\Delta$ and $\sigma = 0.5\Delta$. Offline session lengths are determined implicitly by fixing the number of nodes that are offline at the same time. For the two scenarios 50% or 80% of the nodes were offline at the same time, respectively.

The results of the algorithms in these extreme failure scenarios can be seen in Figure 6.5. As we can see, the different types of failures (whether examined separately or combined together) result only in delay on the convergence rate, but the algorithm still converges. This is in sharp contrast with competing approaches that require synchronization, such as GRADSVD, for example.

An interesting observation is that when only churn is modeled, at the beginning of the simulation convergence is actually faster. This effect is due to the fact that since most of the nodes are offline, the effective network is smaller, but this small sample still allows for an approximation of the SVD. However, convergence eventually slows down as full precision cannot be reached without taking most of the nodes into account in the particular matrix we experiment with.

6.6 Conclusions

In this chapter we proposed an SGD algorithm with an update rule that is based on the GOLF idea and has stable fix points only in the SVD of a matrix A. The output of the algorithm for rank k are two matrices X and Y that contain scaled versions of the first k left and right singular vectors of A, respectively. Matrices X and Y are unique apart from the scaling of the columns.

The most important feature of the algorithms is a P2P sense privacy preservation: we operate over fully distributed data where each network node stores one full row of A as its private data. The output matrix X is also fully distributed: node i that stores row i of A computes row i of X, the private model for the node. Matrices A and X are private in that only the node that stores a given row has access to it. A version of the matrix Y is available in full at all nodes.

Through experimental evaluation we studied the convergence speed of the algorithm, and showed that it is competitive with other gradient methods that require more freedom for data access. We also demonstrated the remarkable robustness of the method in extreme failure scenarios.

Our future work includes addressing interesting challenges such as dynamically changing data, and investigating methods for further improving the efficiency of the protocol, for example, through sampling from the rows and/or columns of A using several sampling techniques.

CHAPTER 7

Summary

The main goal of this thesis was to present a possible way of machine learning on fully distributed data without collecting and storing it in a central location. Here, we expect that a huge number of computational units solve machine learning tasks together and communicate with each other via message passing only. We presented a gossip-based framework to handle this learning problem. Within this framework, various algorithms can be applied. In Chapter 3 we described this framework and possible instantiations of several learning algorithms. Then in the later chapters we presented more sophisticated methods and applications of this framework.

Here we give a short summary of this study, where each section corresponds to a thesis point. At the end of these sections we give an itemized list of the main contributions and the corresponding publications. Later in this chapter we give a summary of our research in Hungarian as well.

7.1 Gossip-Based Machine Learning

We proposed gossip learning as a generic framework used to learn models, based on stochastic gradient search, on fully distributed data in large scale P2P systems. The basic idea behind gossip learning is that many models perform random walks in the network, while being updated at every node they visit. We presented numerous instantiations of gossip-based learning algorithms e.g. Pegasos SVM, Logistic Regression and ANN.

The framework makes it possible to compute predictions locally at every node in the network at any point in time without additional communication cost. Furthermore, it has an acceptable message complexity: each node sends one message in each gossip cycle.

The proposed framework supports privacy preservation, since the data never leaves the node that stores it. The private data can be observed only by sending specific models for a node and monitoring its results.

Main contributions:

- A fully distributed learning framework (GOLF);
- Numerous implemented algorithms;
- The models can be used locally for every node;
- The framework supports privacy preservation;
- The corresponding paper is: [89]

7.2 Fully Distributed Boosting

We demonstrated that the above-described gossip learning framework is suitable for the implementation of a multi-class boosting algorithm. To achieve this, we proposed a modification of the original FILTERBOOST which allows it to learn multi-class models in a purely online way, and we proved theoretically that the resulting algorithm optimizes a suitably defined negative log likelihood measure. The significance of this result is that a state-of-the-art machine learning technique from the point of view of the quality of the learned models is available in fully distributed systems.

We also pointed out the lack of model diversity as a potential problem with GOLF. We provided a solution that is effective in preserving the difference between the best model and the average models. This allowed us to propose spreading the best model as a way of benefitting from the large number of models in the network.

Main contributions:

- An implementation of a fully distributed multi-class boosting method;
- An online version of the FILTERBOOST and the theoretical derivation;
- The model diversity preservation aspect of the GOLF;
- The corresponding paper is: [52]

7.3 Handling Concept Drift

Here, we proposed adaptive versions of our GOLF framework: ADAGOLF and CDDGOLF. With these methods the framework is capable of handling the change in the data patterns that we want to learn. In the case of ADAGOLF, the adaptivity is implemented through the management of the age distribution of the models in the network, ensuring that there is sufficient diversity of different ages in the pool. This method results that in the network there will be young (adaptive) and old (high performance) models. CDDGOLF also restarts some of the models, but this decision is based on the performance history of the model. This method can detect the occurrence of the concept drift as well.

Our main conclusion is that in those scenarios where the sampling rate from the underlying distribution is low relative to the speed of drift, our solutions clearly outperform all the baseline solutions, approximating the "God's Eye view" model, which represents the best possible performance.

We also showed that our algorithms can be enhanced to deal with (or rather, be robust to) higher sample rates as well, although in this case purely local model building can also be sufficient.

Main contributions:

- Two adaptive learning mechanism for GOLF, based on model restarts

  - One of them maintains the age distribution of the models
  - The other resets the models that have low performance;

- Drift handling and detection capabilities;
- High performance on low sampling rate;
- The corresponding papers are: [50, 51, 53]

7.4 Singular Value Decomposition

Here, we proposed an SGD algorithm with an update rule that solves the problem of the low-rank decomposition of a matrix in a fully distributed manner. Additionally, we proposed a modification that has stable fix points only in the SVD of a matrix A. The output of the algorithm for rank k are two matrices X and Y that contain scaled versions of the first k left and right singular vectors of A, respectively. Matrices X and Y are unique apart from the scaling of the columns.

Matrices A and X are private, that is only the node which stores a given row has access to it. A version of the matrix Y is available in full at all nodes.

Through experimental evaluation we studied the convergence speed of the algorithm, and showed that it is competitive with other gradient methods that require more freedom for data access. We also demonstrated the remarkable robustness of the method in extreme failure scenarios.

Main contributions:

- An SGD based matrix low-rank decomposition technique in GOLF;
- A method that converges to a solution that corresponds to the singular value decomposition;
- The node related parts of matrices (the sensitive data) never leave the nodes;
- The corresponding paper is: [49]

Összefoglaló

A tézis fő célkitűzése egy olyan módszer bemutatása, amely segítségével teljesen elosztott adatbázisokon alkalmazhatunk gépi tanuló módszereket anélkül, hogy az adatokat egy központi helyre összegyűjtenénk. Az elvárásunk az, hogy a nagyszámú számítástechnikai eszköz együttműködve oldja meg a gépi tanulási problémákat kizárólag üzenetküldések segítségével. Bemutattunk egy pletyka-alapú keretrendszert, amely képes megoldani a fent említett problémát. A harmadik fejezetben részletesen bemutattuk e keretrendszert, valamint számos, a keretrendszerben alkalmazható tanuló algoritmust. A későbbi fejezetekben pedig fejlettebb módszereket és a keretrendszer alkalmazhatóságát vizsgáltuk.

Ebben a fejezetben adunk egy összefoglalót a disszertációról, melynek az alfejezetei az egyes tézispontokhoz tartozó eredményeket mutatják be. Ezen alfejezetek végén pontokba szedve kiemeljük az elért eredményeket, valamint a kapcsolódó publikációkat.

7.5. Pletykaalapú gépi tanulás

Bemutattunk egy pletykaalapú általános keretrendszert, amely gépi tanuló modellek sztochasztikus gradiens alapú tanítását teszi lehetővé P2P rendszerekben, teljesen elosztott adatok felett. A pletykaalapú tanulás alapötlete, hogy modellek vándorolnak a hálózat elemein véletlen sétát leírva, miközben az eszközök frissítik a fogadott modelleket. A fejezetben bemutattunk számos gépi tanuló algoritmust, amelyek beilleszthetők ebbe a keretbe, mint például: Pegasos SVM, Logistic Regression, ANN.

A keretrendszer lehetővé teszi a hálózati csomópontok számára a predikció lokális kiszámítását, bármilyen üzenet küldése nélkül. Továbbá a kommunikációs költsége is kielégítő, csomópontonként egy-egy üzenet elküldés történik minden ciklusban.

Azáltal, hogy a lokális adat sosem hagyja el az eszközt, amely tárolja azt, a keretrendszer támogatja az érzékeny adatok védelmét. Ezen adatok csak speciálisan kialakított modellek segítségével figyelhetők meg.

Fő eredmények:

- Teljesen elosztott tanuló keretrendszer (GoLF);
- Számos megvalósított tanuló algoritmus;
- Lokálisan használható modellek;
- Érzékeny adatok védelmének támogatása;
- Kapcsolódó publikáció: [89]

7.6. Teljesen elosztott turbózás

Bemutattuk, hogy a fenti keretrendszer alkalmas a többosztályos turbózás megvalósítására is. Ennek elérése érdekében bemutattunk egy módosított FilterBoost algoritmust, amely így teljesen online módon képes többosztályos modellek tanítására. Elméletileg igazoltuk, hogy a definiált módszer optimalizálja a megadott negatív log-likelihood mértéket. Az eredmények jelentősége, hogy a gépi tanuló modell minősége szempontjából fejlett algoritmusok is alkalmazhatók teljesen elosztott rendszerekben.

Továbbá rámutattunk, hogy a modellek diverzitásának csökkenése lehetséges probléma a GoLF keretrendszerben. Adtunk egy megoldást, amely segítségével fenntartható a hálózatban jelen lévő modellek változatossága. Ez lehetővé teszi a legjobb modellek elterjesztését az eszközökön, kihasználva a hálózatban rendelkezésre álló modellek számát.

Fő eredmények:

- Elosztott többosztályos turbózás megvalósítása;
- Online FILTERBOOST algoritmus és elméleti levezetése;
- A modellek változatosságának fenntartása a GOLF keretrendszerben;
- Kapcsolódó publikáció: [52]

7.7. Fogalomsodródás kezelése

Bemutattuk a GOLF keretrendszer két adaptív változatát, melyek az ADAGOLF és a CDDGOLF. Ezekkel a módszerekkel a keretrendszer alkalmas az adatokban rejlő azon minták változásának kezelésére, amelyeket meg akarunk tanulni. Az ADAGOLF esetén az adaptivitást a hálózatban jelen lévő modellek életkoreloszlásának fenntartásával értük el, figyelve a megfelelő változatosság fenntartására az életkorokban. Így a hálózatban lesznek fiatal (adaptív) és idős (jó teljesítményű) modellek. A CDDGOLF szintén a modellek újraindításával éri el az adaptivitást, viszont a döntés itt a modell teljesítményének az előzményétől függ. Ezen módszer képes jelezni az eloszlás változását is.

A legfőbb eredményünk azon esetekhez köthető, amikor az eloszlás mintázásának a sebessége viszonylag alacsony az eloszlás változásának a sebességéhez képest. Ebben az esetben a módszerünk teljesítménye jelentősen felülmúlja az alap algoritmusok eredményeit és megközelíti az elérhető legjobb teljesítményt.

Továbbá bemutattuk, hogy nagyobb mintavételezési sebesség esetén is alkalmazható a módszerünk, habár ebben az esetben a lokális adaton tanított modell is kielégítő eredményt ér el.

Fő eredmények:

- Kétféle adaptív módszer a GOLF keretrendszerhez, melyek
  - Egyike fenntartja a modellek életkorának eloszlását

  - A másik pedig újraindítja az alacsony teljesítményű modelleket;

- Eloszlás változásának követése, valamint detektálása;
- Kiemelkedő teljesítmény az eloszlás ritka mintavételezése esetén;
- Kapcsolódó publikációk: [50, 51, 53]

7.8. Szinguláris felbontás

Bemutattunk egy SGD alapú módszert, amely segítségével alacsony rangú mátrixfaktorizáció feladata oldható meg, teljesen elosztott környezetben. Továbbá a módszer egy módosítását, amely egy A mátrix SVD felbontásához konvergál. Az algoritmus eredménye k rang esetén azon X és Y mátrixok, amelyek az A mátrix első k jobb és bal szinguláris vektorainak skálázott változatát tartalmazzák. Az X és Y mátrixok egyediek, eltekintve az oszlopaik hosszától.

Az A és X mátrixok sorait csak azok az eszközök érik el, amelyek tárolják azt. Az Y mátrix egy-egy példánya pedig lokálisan elérhető a hálózat összes eleme számára.

A tapasztalati kiértékelések alapján vizsgáltuk az algoritmus konvergenciájának sebességét, és bemutattuk, hogy versenyképes más gradiens alapú módszerekhez képest, amelyek ráadásul extra adathozzáférést igényelnek. Továbbá a módszer hálózati hibatűrő képessége is figyelemre méltó.

Fő eredmények:

- SGD alapú alacsony rangú matrixfelbontó módszer a GoLF keretben;
- Egy módszer, amely az SVD eredményéhez konvergál;
- Az érzékeny adatok nem hagyják el az eszközöket;
- Kapcsolódó publikáció: [49]

Acknowledgements

This thesis could not have been written without the support and inspiration of numerous people. Here I would like to thank all of them.

Fist of all, I would like to thank my supervisor Márk Jelasity for his guidance, for supporting my work with useful comments and letting me work at the Research Group on Artificial Intelligence.

I would like to say thank you to my colleagues and friends who helped me discovering interesting fields of science, which allowed me to find my own path. They are listed here in alphabetic order: András Bánhalmi, Róbert Busa-Fekete, Richárd Farkas, Róbert Ormándi and György Szarvas. Moreover, I would like to thank Veronika Vincze for correcting this thesis from a linguistic point of view.

I would also like to thank to Emese Varga for her support and inspiration. Last, but not least I wish to thank my parents for their love, support and for believing in me.

References

[1] Filelist. `http://www.filelist.org`, 2005.

[2] Tarek Abdelzaher, Yaw Anokwa, Peter Boda, Jeff Burke, Deborah Estrin, Leonidas Guibas, Aman Kansal, Samuel Madden, and Jim Reich. Mobiscopes for human spaces. *IEEE Pervasive Computing*, 6(2):20–29, April 2007.

[3] Dimitris Achlioptas and Frank McSherry. On spectral learning of mixtures of distributions. In *Proc. 18th Annual Conference on Learning Theory (COLT)*, pages 458–469, 2005.

[4] Ethem Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2nd edition, 2010.

[5] Hock Ang, Vivekanand Gopalkrishnan, Steven Hoi, and Wee Ng. Cascade RSVM in peer-to-peer networks. In Walter Daelemans, Bart Goethals, and Katharina Morik, editors, *Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, volume 5211 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2008.

[6] Hock Ang, Vivekanand Gopalkrishnan, Wee Ng, and Steven Hoi. Communication-efficient classification in P2P networks. In Wray Buntine, Marko Grobelnik, Dunja Mladenic, and John Shawe-Taylor, editors, *Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, volume 5781 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2009.

[7] Hock Ang, Vivekanand Gopalkrishnan, Wee Ng, and Steven Hoi. On classifying drifting concepts in P2P networks. In José Balcázar, Francesco Bonchi, Aristides Gionis, and Michèle Sebag, editors, *Machine Learning and Knowledge Discovery in*

Databases (ECML PKDD), volume 6321 of *Lecture Notes in Computer Science*, pages 24–39. Springer, 2010.

[8] Árpád Berta, István Hegedűs, and Márk Jelasity. Dimension reduction methods for collaborative mobile gossip learning. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 393–397, Feb 2016.

[9] Arthur U. Asuncion, Padhraic Smyth, and Max Welling. Asynchronous distributed estimation of topic models for document analysis. *Statistical Methodology*, 8(1):3 – 17, 2011.

[10] Yossi Azar, Amos Fiat, Anna R. Karlin, Frank McSherry, and Jared Saia. Spectral analysis of data. In *Proc. 33rd Symposium on Theory of Computing (STOC)*, pages 619–626, 2001.

[11] Boris Babenko, Ming-Hsuan Yang, and Serge Belongie. A family of online boosting algorithms. In *Computer Vision Workshops (ICCV Workshops)*, pages 1346–1353, 2009.

[12] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[13] Manuel Baena-García, José del Campo-Ávila, Raúl Fidalgo, Albert Bifet, Ricard Gavaldá, and Rafael Morales-Bueno. Early drift detection method. In *Fourth International Workshop on Knowledge Discovery from Data Streams*, volume 6, pages 77–86, 2006.

[14] Arno Bakker, Elth Ogston, and Maarten van Steen. Collaborative filtering using random neighbours in peer-to-peer networks. In *Proceeding of the 1st ACM international workshop on Complex networks meet information and knowledge management (CNIKM '09)*, pages 67–75, New York, NY, USA, 2009. ACM.

[15] Ron Bekkerman, Mikhail Bilenko, and John Langford, editors. *Scaling up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press, December 2011.

[16] Austin R. Benson, David F. Gleich, and James Demmel. Direct qr factorizations for tall-and-skinny matrices in mapreduce architectures. *CoRR*, 2013.

[17] Michael W. Berry, Susan T. Dumais, and Gavin W. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37(4):573–595, 1995.

[18] Árpád Berta, István Hegedűs, and Róbert Ormándi. Lightning fast asynchronous distributed k-means clustering. In *22th European Symposium on Artificial Neural Networks*, ESANN 2014, pages 99–104, 2014.

[19] Kanishka Bhaduri, Ran Wolff, Chris Giannella, and Hillol Kargupta. Distributed decision-tree induction in peer-to-peer systems. *Stat. Anal. Data Min.*, 1:85–103, June 2008.

[20] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[21] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In Yves Lechevallier and Gilbert Saporta, editors, *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187, Paris, France, August 2010. Springer.

[22] Joseph K. Bradley and Robert E. Schapire. FilterBoost: Regression and classification on large datasets. In *Advances in Neural Information Processing Systems*, volume 20. The MIT Press, 2008.

[23] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19 (NIPS 2006)*, pages 281–288. MIT Press, 2007.

[24] Fan Chung, Linyuan Lu, and Van Vu. Eigenvalues of random power law graphs. *Annals of Combinatorics*, 7(1):21–33, 2003.

[25] M. Collins, R.E. Schapire, and Y. Singer. Logistic regression, AdaBoost and Bregman distances. *Machine Learning*, 48:253–285, 2002.

[26] Terence Craig and Mary E. Ludloff. *Privacy and Big Data*. O'Reilly Media, September 2011.

[27] Nello Cristianini and John Shawe-Taylor. *An introduction to Support Vector Machines and other kernel-based learning methods*. Cambridge University Press, 2000.

[28] Souptik Datta, Kanishka Bhaduri, Chris Giannella, Ran Wolff, and Hillol Kargupta. Distributed data mining in peer-to-peer networks. *IEEE Internet Comp.*, 10(4):18–26, July 2006.

[29] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[30] Sarah Jane Delany, Pádraig Cunningham, Alexey Tsymbal, and Lorcan Coyle. A case-based technique for tracking concept drift in spam filtering. *Know.-Based Syst.*, 18(4-5):187–195, August 2005.

[31] Anton Dries and Ulrich Rückert. Adaptive concept drift detection. *Stat. Anal. Data Min.*, 2(56):311–327, December 2009.

[32] Petros Drineas, Alan Frieze, Ravi Kannan, Santosh Vempala, and V. Vinay. Clustering large graphs via the singular value decomposition. *Machine Learning*, pages 9–33, 2004.

[33] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. Fast monte carlo algo-

rithms for matrices ii: Computing a low-rank approximation to a matrix. *SIAM J. Comput.*, 36(1):158–183, 2006.

[34] Petros Drineas, Iordanis Kerenidis, and Prabhakar Raghavan. Competitive recommendation systems. In *Proc. 34th Symposium on Theory of Computing (STOC)*, pages 82–90, 2002.

[35] Wei Fan, Salvatore J. Stolfo, and Junxin Zhang. The application of AdaBoost for distributed, scalable and on-line learning. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, pages 362–366, New York, NY, USA, 1999. ACM.

[36] Tom Fawcett. An introduction to ROC analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[37] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on Machine Learning (ICML 1996)*, pages 148–156, 1996.

[38] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. of Comp. and Syst. Sci.*, 55:119–139, 1997.

[39] Jerome H. Friedman. Stochastic gradient boosting. *Comput. Stat. Data Anal.*, 38(4):367–378, February 2002.

[40] Joao Gama, Pedro Medas, Gladys Castillo, and Pedro Rodrigues. Learning with drift detection. In *Advances in Artificial Intelligence, Proceedings of SBIA 2004*, volume 3171 of *LNCS*, pages 286–295. Springer, 2004.

[41] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proc. 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 69–77. ACM, 2011.

[42] Alan Genz. Methods for generating random orthogonal matrices. In H. Niederreiter and J. Spanier, editors, *Proc. Monte Carlo and Quasi-Monte Carlo Methods (MC-QMC 1998)*, pages 199–213. Springer, 1999.

[43] Genevieve Gorrell. Generalized Hebbian algorithm for incremental singular value decomposition in natural language processing. In Diana McCarthy and Shuly Wintner, editors, *Proc. 11th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*. The Association for Computer Linguistics, 2006.

[44] Geoffrey Grimmett and David Stirzaker. *Probability and Random Processes*. Texts from Oxford University Press. Oxford University Press, 2001.

[45] Isabelle Guyon, Asa Ben Hur, Steve Gunn, and Gideon Dror. Result analysis of the

nips 2003 feature selection challenge. In *Advances in Neural Information Processing Systems 17*, pages 545–552. MIT Press, 2004.

[46] Peng Han, Bo Xie, Fan Yang, Jiajun Wang, and Ruimin Shen. A novel distributed collaborative filtering algorithm and its implementation on P2P overlay network. In Honghua Dai, Ramakrishnan Srikant, and Chengqi Zhang, editors, *Advances in Knowledge Discovery and Data Mining*, volume 3056 of *LNCS*, pages 106–115. Springer, 2004.

[47] István Hegedűs, Árpád Berta, Levente Kocsis, András A. Benczúr, and Márk Jelasity. Robust decentralized low-rank matrix decomposition. *ACM Trans. Intell. Syst. Technol.*, 7(4):62:1–62:24, May 2016.

[48] István Hegedűs and Márk Jelasity. Distributed differentially private stochastic gradient descent: An empirical study. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 566–573, Feb 2016.

[49] István Hegedűs, Márk Jelasity, Levente Kocsis, and András A. Benczúr. Fully distributed robust singular value decomposition. In *Proceedings of the 14th IEEE Fourteenth International Conference on Peer-to-Peer Computing (P2P)*, P2P'14. IEEE, 2014.

[50] István Hegedűs, Lehel Nyers, and Róbert Ormándi. Detecting concept drift in fully distributed environments. In *2012 IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics*, SISY'12, pages 183–188. IEEE, 2012.

[51] István Hegedűs, Róbert Ormándi, and Márk Jelasity. Massively distributed concept drift handling in large networks. *Advances in Complex Systems*, 16(4&5):1350021, 2013.

[52] István Hegedűs, Busa-Fekete Róbert, Ormándi Róbert, Jelasity Márk, and Kégl Balázs. Peer-to-peer multi-class boosting. In Christos Kaklamanis, Theodore Papatheodorou, and Paul Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 389–400. Springer Berlin / Heidelberg, 2012.

[53] István Hegedűs, Ormándi Róbert, and Jelasity Márk. Gossip-based learning under drifting concepts in fully distributed networks. In *2012 IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems*, SASO'12, pages 79–88. IEEE, 2012.

[54] Chase Hensel and Haimonti Dutta. GADGET SVM: a gossip-based sub-gradient svm solver. In *International Conference on Machine Learning (ICML), Numerical Mathematics in Machine Learning Workshop*, 2009.

[55] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '01, pages 97–106, New York, NY, USA, 2001. ACM.

[56] Sibren Isaacman, Stratis Ioannidis, Augustin Chaintreau, and Margaret Martonosi. Distributed rating prediction in user generated content streams. In *Proc. Fifth ACM Conf. on Rec. Sys.*, pages 69–76. ACM, 2011.

[57] Márk Jelasity, Geoffrey Canright, and Kenth Engø-Monsen. Asynchronous distributed power iteration with gossip-based normalization. In *Euro-Par 2007*, volume 4641 of *LNCS*, pages 514–525. Springer, 2007.

[58] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. on Computer Systems*, 23(3):219–252, August 2005.

[59] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-Man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321–2339, 2009.

[60] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3):8, August 2007.

[61] Ravindran Kannan, Hadi Salmasian, and Santosh Vempala. The spectral method for general mixture models. In *Proc. 18th Annual Conference on Learning Theory (COLT)*, pages 444–457, 2005.

[62] Balázs Kégl and Róbert Busa-Fekete. Boosting products of base classifiers. In *Intl. Conf. on Machine Learning*, volume 26, pages 497–504, Montreal, Canada, 2009.

[63] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *Proc. 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03)*, pages 482–491. IEEE Computer Society, 2003.

[64] David Kempe and Frank McSherry. A decentralized algorithm for spectral analysis. In *Proc. 36th Symposium on Theory of Computing (STOC)*, pages 561–568. ACM, 2004.

[65] Jon Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[66] J. Zico Kolter and Marcus A. Maloof. Dynamic weighted majority: An ensemble method for drifting concepts. *J. Mach. Learn. Res.*, 8:2755–2790, December 2007.

[67] Satish Babu Korada, Andrea Montanari, and Sewoong Oh. Gossip PCA. In *Proc. ACM SIGMETRICS joint int. conf. on Measurement and modeling of comp. sys.*, pages 209–220. ACM, 2011.

[68] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[69] Wojtek Kowalczyk and Nikos Vlassis. Newscast EM. In *17th Advances in Neural Information Processing Systems (NIPS)*, pages 713–720, Cambridge, MA, 2005. MIT Press.

[70] N.D. Lane, E. Miluzzo, Hong Lu, D. Peebles, T. Choudhury, and A.T. Campbell. A survey of mobile phone sensing. *Communications Magazine, IEEE*, 48(9):140–150, September 2010.

[71] Terran Lane and Carla E. Brodley. Approaches to online learning and concept drift for user identification in computer security. In *Proceedings of the 4th International Conference on Knowledge Discovery and Datamining*, 1998.

[72] Quoc Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg Corrado, Jeff Dean, and Andrew Ng. Building high-level features using large scale unsupervised learning. In John Langford and Joelle Pineau, editors, *Proc. 29th International Conference on Machine Learning (ICML)*, pages 81–88. Omnipress, 2012.

[73] Yongjun Liao, Pierre Geurts, and Guy Leduc. Network distance prediction based on decentralized matrix factorization. In Mark Crovella, LauraMarie Feeney, Dan Rubenstein, and S.V. Raghavan, editors, *Proc. 9th Int. IFIP TC 6 Netw. Conf.*, volume 6091 of *LNCS*, pages 15–26. Springer, 2010.

[74] Qing Ling, Yangyang Xu, Wotao Yin, and Zaiwen Wen. Decentralized low-rank matrix completion. In *Proceedings of the 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2925–2928, March 2012.

[75] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conf. on Uncertainty in Artif. Intel.*, 2010.

[76] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[77] Ping Luo, Hui Xiong, Kevin Lü, and Zhongzhi Shi. Distributed classification in peer-to-peer networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 968–976, New York, NY, USA, 2007. ACM.

[78] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. Identifying suspicious URLs: an application of large-scale online learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML '09)*, pages 681–688, New York, NY, USA, 2009. ACM.

[79] David A. McGrew and Scott R. Fluhrer. Multiple forgery attacks against message authentication codes. *IACR Cryptology ePrint Archive*, 2005:161, 2005.

[80] Frank McSherry. Spectral partitioning of random graphs. In *Proc. 42nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 529–537, 2001.

[81] Milena Mihail and Christos Papadimitriou. On the eigenvalue power law. In José D.P. Rolim and Salil Vadhan, editors, *Rand. and Approx. Tech. in Comp. Sci.*, volume 2483 of *LNCS*, pages 254–262. Springer, 2002.

[82] Leandro L Minku, Allan P White, and Xin Yao. The impact of diversity on online ensemble learning in the presence of concept drift. *Knowledge and Data Engineering, IEEE Transactions on*, 22(5):730 –742, May 2010.

[83] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 2 edition, 1997.

[84] Alberto Montresor and Márk Jelasity. Peersim: A scalable P2P simulator. In *Proc. 9th IEEE Int. Conf. on Peer-to-Peer Comp.*, pages 99–100. IEEE, 2009. extended abstract.

[85] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving matrix factorization. In *Proc. 2013 ACM SIGSAC Conf. on Comp. and Comm. Security (CCS'13)*, pages 801–812. ACM, 2013.

[86] T. Nis. JAMA: A Java matrix package, 1999.

[87] Róbert Ormándi, István Hegedűs, and Márk Jelasity. Overlay management for fully distributed user-based collaborative filtering. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, pages 446–457, Berlin, Heidelberg, 2010. Springer-Verlag.

[88] Róbert Ormándi, István Hegedűs, and Márk Jelasity. Asynchronous peer-to-peer data mining with stochastic gradient descent. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 528–540, Berlin, Heidelberg, 2011. Springer-Verlag.

[89] Róbert Ormándi, István Hegedűs, and Márk Jelasity. Gossip learning with linear models on fully distributed data. *Concurrency and Computation: Practice and Experience*, 25(4):556–571, 2013.

[90] N.C. Oza and S. Russell. Online bagging and boosting. In *Proc. Eighth Intl. Workshop on Artificial Intelligence and Statistics*, 2001.

[91] Christos H. Papadimitriou, Hisao Tamaki, Prabhakar Raghavan, and Santosh Vempala. Latent semantic indexing: A probabilistic analysis. *Journal of Computer and System Sciences*, 61(2):217–235, 2000.

[92] Byung-Hoon Park and Hillol Kargupta. Distributed data mining: Algorithms, systems, and applications. In Nong Ye, editor, *The Handbook of Data Mining*. CRC Press, 2003.

[93] Alex (Sandy) Pentland. Society's nervous system: Building effective government, energy, and public health systems. *Computer*, 45(1):31–38, January 2012.

[94] John R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.

[95] Herbert Robbins and Sutton Monro. A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407, 09 1951.

[96] Thomas Rodenhausen, Mojisola Anjorin, Renato Domínguez García, and Christoph Rensing. Context determines content: an approach to resource recommendation in folksonomies. In *Proceedings of the 4th ACM RecSys workshop on Recommender systems and the social web*, RSWeb '12, pages 17–24, New York, NY, USA, 2012. ACM.

[97] Jelle Roozenburg. Secure decentralized swarm discovery in Tribler. Master's thesis, Parallel and Distributed Systems Group, Delft University of Technology, 2006.

[98] Frank Rosenblatt. *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Report (Cornell Aeronautical Laboratory). Spartan Books, 1962.

[99] Robert E. Schapire and Yoram Singer. Improved boosting algorithms using confidence-rated predictions. *Machine Learning*, 37(3):297–336, 1999.

[100] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. Pegasos: primal estimated sub-gradient solver for SVM. *Mathematical Programming B*, 2010.

[101] Stefan Siersdorfer and Sergej Sizov. Automatic document organization in a P2P environment. In *Advances in Information Retrieval*, volume 3936 of *LNCS*, pages 265–276. Springer, 2006.

[102] Nathan Srebro and Tommi Jaakkola. Weighted low-rank approximations. In *Proc. 20th International Conference on Machine Learning (ICML)*, pages 720–727. AAAI Press, 2003.

[103] Stephen V Stehman. Selecting and interpreting measures of thematic classification accuracy. *Remote sensing of Environment*, 62(1):77–89, 1997.

[104] W. Nick Street and YongSeog Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '01, pages 377–382, New York, NY, USA, 2001. ACM.

[105] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proc. 6th ACM SIGCOMM conf. on Internet Measurement (IMC)*, pages 189–202. ACM, 2006.

[106] Balázs Szörényi, Róbert Busa-Fekete, István Hegedűs, Ormándi Róbert, Jelasity

Márk, and Kégl Balázs. Gossip-based distributed stochastic bandit algorithms. In *Proceedings of The 30th International Conference on Machine Learning*, volume 28(3) of *ICML'13*, page 19–27. JMLR Workshop and Conference Proceedings, 2013.

[107] Norbert Tölgyesi and Márk Jelasity. Adaptive peer sampling with Newscast. In *Euro-Par 2009*, volume 5704 of *LNCS*, pages 523–534. Springer, 2009.

[108] Amund Tveit. Peer-to-peer based recommendations for mobile commerce. In *Proc. 1st Intl. workshop on Mobile commerce (WMC '01)*, pages 26–29. ACM, 2001.

[109] Robbert van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, May 2003.

[110] Jilles Vreeken, Matthijs van Leeuwen, and Arno Siebes. Characterising the difference. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '07, pages 765–774, New York, NY, USA, 2007. ACM.

[111] Haixun Wang, Wei Fan, Philip S. Yu, and Jiawei Han. Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '03, pages 226–235, New York, NY, USA, 2003. ACM.

[112] Geoffrey I. Webb, Michael J. Pazzani, and Daniel Billsus. Machine learning for user modeling. *User Modeling and User-Adapted Interaction*, 11(1-2):19–29, March 2001.

[113] Guiying Wei, Tao Zhang, Sen Wu, and Lei Zou. An ensemble classifier method for classifying data streams with recurrent concept drift. In *Proceedings of the 4th International Conference on Awareness Science and Technology (iCAST)*, pages 3–9, 2012.

[114] Bernard Widrow and Marcian E. Hoff. Adaptive Switching Circuits. In *1960 IRE WESCON Convention Record*, volume 4, pages 96–104, 1960.

[115] Martin A. Zinkevich, Alex Smola, Markus Weimer, and Lihong Li. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems 23 (NIPS 2010)*, pages 2595–2603, 2010.

[116] Indre Zliobaite. Learning under concept drift: an overview. Technical Report 1010.4784, arxiv.org, 2010.