QLectives – Socially Intelligent Systems for Quality
Project no. 231200

Instrument: Large-scale integrating project (IP)
Programme: FP7-ICT

Deliverable D2.3.1

Report on the deployment of protocols in living lab environments

Submission date: 2012-03-01

Start date of project: 2009-03-01 Duration: 48 months

Organisation name of lead contractor for this deliverable: USZ

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Document information

1.1 Authors

Author	Organisation	E-mail
Márk Jelasity	USZ	jelasity@inf.u-szeged.hu
Kornél Csernai	USZ	csko@inf.u-szeged.hu
István Hegedűs	USZ	ihegedus@inf.u-szeged.hu
Róbert Ormándi	USZ	ormandi@inf.u-szeged.hu

1.2 Other contributors

Name	Organisation	E-mail
Boudewijn Schoon	TUD	p.b.schoon@tudelft.nl
Niels Zeilemaker	TUD	n.zeilemaker@tudelft.nl
Johan Pouwelse	TUD	peer2peer@gmail.com

1.3 Document history

Version#	Date	Change
V1.2	1 February 2012	Final version
V1.1	13 January 2012	Minor reorganization of section structure
V1.0	7 January 2012	First complete draft (internal)
V0.1	16 December 2011	First draft internal consortium version

1.4 Document data

Keywords	QLectives, peer-to-peer, machine learning, gossip, quality metadata, data mining, QMedia
Editor address data	jelasity@inf.u-szeged.hu
Delivery date	17 February, 2011

1.5 Distribution list

Date	Issue	E-mail
	Consortium members	QLECTIVES@list.surrey.ac.uk
	Project officer	Jose.FERNANDEZ-VILLACANAS@ec.europa.eu
	EC archive	INFSO-ICT-231200@ec.europa.eu

QLectives Consortium

This document is part of a research project funded by the ICT Programme of the Commission of the European Communities as grant number ICT-2009-231200.

University of Surrey (Coordinator)
Department of Sociology/Centre for Research in Social Simulation
Guildford GU2 7XH
Surrey
United Kingdom
Contact person: Prof. Nigel Gilbert
E-mail: n.gilbert@surrey.ac.uk

University of Fribourg
Department of Physics
Fribourg 1700
Switzerland
Contact person: Prof. Yi-Cheng Zhang
E-mail: yi-cheng.zhang@unifr.ch

Technical University of Delft
Department of Software Technology
Delft, 2628 CN
Netherlands
Contact Person: Dr Johan Pouwelse
E-mail: j.a.pouwelse@tudelft.nl

University of Warsaw
Faculty of Psychology
Warsaw 00927
Poland
Contact Person: Prof. Andrzej Nowak
E-mail: nowak@fau.edu

ETH Zurich
Chair of Sociology, in particular Modelling and Simulation
Zurich, CH-8092
Switzerland
Contact person: Prof. Dirk Helbing
E-mail: dhelbing@ethz.ch

Centre National de la Recherche Scientifique, CNRS
Paris 75006,
France
Contact person : Dr. Camille ROTH
E-mail: camille.roth@polytechnique.edu

University of Szeged
MTA-SZTE Research Group on Artificial Intelligence
Szeged 6720, Hungary
Contact person: Dr Mark Jelasity
E-mail: jelasity@inf.u-szeged.hu

Institut für Rundfunktechnik GmbH
Munich 80939
Germany
Contact person: Dr. Christoph Dosch
E-mail: dosch@irt.de

QLectives introduction

QLectives is a project bringing together top social modelers, peer-to-peer engineers and physicists to design and deploy next generation self-organising socially intelligent information systems. The project aims to combine three recent trends within information systems:

- **Social networks** - in which people link to others over the Internet to gain value and facilitate collaboration
- **Peer production** - in which people collectively produce informational products and experiences without traditional hierarchies or market incentives
- **Peer-to-Peer systems** - in which software clients running on user machines distribute media and other information without a central server or administrative control

QLectives aims to bring these together to form Quality Collectives, i.e. functional decentralised communities that self-organise and self-maintain for the benefit of the people who comprise them. We aim to generate theory at the social level, design algorithms and deploy prototypes targeted towards two application domains:

- **QMedia** - an interactive peer-to-peer media distribution system (including live streaming), providing fully distributed social filtering and recommendation for quality
- **QScience** - a distributed platform for scientists allowing them to locate or form new communities and quality reviewing mechanisms, which are transparent and promote quality

The approach of the QLectives project is unique in that it brings together a highly inter-disciplinary team applied to specific real world problems. The project applies a scientific approach to research by formulating theories, applying them to real systems and then performing detailed measurements of system and user behaviour to validate or modify our theories if necessary. The two applications will be based on two existing user communities comprising several thousand people - so-called "Living labs", media sharing community tribler.org; and the scientific collaboration forum EconoPhysics.

Executive summary

In WP2.3 our task was to design and implement several algorithms that fall in the data mining domain. These algorithms had to be fully distributed, robust, and efficient, and they were required to be implemented in the P2P QLectives living lab platform. We achieved these goals by proposing and implementing the *gossip learning framework* (GOLF), which is a generic scheme for applying data mining algorithms in unreliable distributed environments. To instantiate the scheme for a specific data mining algorithm, one needs an online version of the algorithm, along with an optional (but recommended) combination scheme that is elaborated on below.

We first define our system model to be the classical P2P model, where nodes can have overlay links (defined by the "knows about" relation), over which they can send messages. Communication is unreliable and asynchronous. We assume that there is a huge number of nodes, and data is fully distributed over these nodes. Each nodes contains a potentially very small proportion of the data, perhaps only a single data record (profile, preferences, personal GPS track, and so on). We assume that due to privacy, data cannot be moved out of a node, and only anonymous information can be communicated.

Our idea to implement data mining algorithms under these assumptions relies on gossip communication, where nodes periodically communicate with random peers. Gossip is often used to spread information or perform global computations. In the case of data mining, what is gossiped are machine learning models: linear model parameters, neural networks, decision trees, and so on. Each node updates the models it receives using its locally stored data using an appropriate online algorithm. The node can also combine models that pass through it, hence speeding up convergence. Afterwards the node gossips these models to random peers. As a result, the models in the network converge to a high quality model that describes the dataset spread over the network.

We studied the case of linear models in detail, both theoretically and experimentally. Linear models are very simple, yet in high dimensional spaces, or with an appropriate non-linear transformation of the feature space, they can be very competitive. In the case of linear models we proposed a model-combination method that is simply the averaging of the models. We argued that performing prediction using the average of two linear models is very close to a weighted vote of the two original models. This means that a repeated averaging-based combination during gossip is a very efficient implementation of weighted voting over an exponentially increasing number of models (which are not independent, however). We demonstrated experimentally that the algorithm is robust and has a favorable convergence speed. Finally, we proved theoretically that the averaging-based combination technique will preserve the convergence of an online SVM learner called Pegasos.

GOLF was implemented for the P2P QLectives platform that is based on Tribler, a

social-based BitTorrent client developed at TU Delft. We used the stable release (version 5.4.x at the time of the implementation) and extended it with our machine learning implementation. GOLF is implemented as a Dispersy community, a feature of the QLectives platform that was introduced during the project. We tested our implementation using a setup of 90 Tribler clients that were run on our high performance server on different ports. In our tests we used a simple perceptron learner over a benchmark database, and validated our results using PeerSim simulations for the same problem.

We mention two possible applications of GOLF. The first is the implementation of recommender systems. We outline a collaborative filtering approach that can be implemented in an online fashion, and hence can serve as a basis for a GOLF implementation. The second is related to WP4.4, where quality metadata needs to be identified and promoted. Our framework can be used to implement data mining algorithms to filter spam, or to detect vandalism in a distributed way. We will report on the results of this application in Deliverable D4.4.4 due in month 48.

We conclude that the implementation of the learning framework in the P2P QLectives platform has been tested and validated, and it is ready for deployment. Since it is a generic platform, it can support any algorithms that have an online variant with a bounded space-complexity of the learned models.

Contents

Chapter 1

Introduction

This document reports on the outcome of WP2.3 and its integration into the P2P QLectives platform. The main goal of WP2.3 was to design and develop several data mining algorithms for ranking and recommendation. The main design goal was to make sure that the algorithms are suitable for a dynamic and large scale P2P environment, and they support privacy preservation through not collecting user data at all but processing it only locally.

In the first years of WP2.3, we have developed a number of algorithms initially for P2P recommendation [34, 36]. During that time, WP2.3 was in close contact with Stream 1 partners who also proposed a number of inspiring algorithms for the same purpose based on a number of physical analogies [33, 43, 52]. The algorithms resulting from Stream 1 were carefully analyzed from the point of view of practical aspects related to P2P environments (scalability, communication constraints, etc) and our objective to analyze the data in situ, without collecting it centrally.

During the first years of the project it also became clear, that Stream 4—in particular WP4.4 for developing self-maintaining mechanisms to provide quality metadata in our QLectives platform—also requires machine learning algorithms, for example, spam filtering or vandalism detection. After realizing this link with Stream 4 we devoted appropriate effort to vandalism detection [35] and we studied the possibilities to allow these algorithms to be supported by the software platform being designed and implemented in WP2.3.

Based on this initial work and the experiences we obtained during the first years, we designed a fully distributed *generic machine learning framework* that is based on gossip, and can be incorporated into the architecture of the P2P QLectives platform being developed in Stream 4. This gossip learning framework (GOLF) was designed to support any machine learning application, including spam filtering, vandalism detection, recommender systems, and so on. The framework can be instantiated by implementing an online learning algorithm, that has a sub-linear model size in the number of examples visited. An optional (but highly recommended) merging algorithm that is able to combine two models into one in a meaningful way can also be implemented, which results in a dramatically increased scalability and speedup.

In Chapter 2 we describe the GOLF framework, and one instantiation based on a linear SVM learner. After, we present theoretical and experimental results with this instantiation.

The implementation of GOLF in our P2P QLectives platform has been completed, and is described in Chapter 3, where we also briefly outline an approach that can be applied to implement a recommender system in GOLF.

GOLF was integrated into the platform as a generic component with access to local user data such as ratings, as well as public global data that is propagated by the platform. An application related to WP4.4 (spam detection) has also been implemented and is ready for testing. This implementation will form the basis for D4.4.4 in the final year.

Chapter 2

Gossip Learning with Linear Models on Fully Distributed Data

Machine learning over fully distributed data poses an important problem in peer-to-peer (P2P) applications. In this model we have one data record at each network node, but without the possibility to move raw data due to privacy considerations. For example, it might be user profiles, ratings, history or sensor readings. This problem is difficult, because there is no possibility to learn local models, the system model offers almost no guarantees for reliability, yet the communication costs need to be kept low. Here we propose gossip learning, a generic approach that is based on multiple models taking random walks over the network in parallel, while applying an online learning algorithm to improve themselves, and getting combined via ensemble learning methods. We present an instantiation of this approach for the case of classification with linear models. Our main contribution is an ensemble learning method which—through the continuous combination of the models in the network—implements a virtual weighted voting mechanism over an exponential number of models at practically no extra cost as compared to independent random walks. We theoretically prove the convergence of the method and perform extensive experiments on benchmark datasets. Our experimental analysis demonstrates the performance and robustness of the proposed approach.

2.1 Introduction

The main attraction of peer-to-peer (P2P) technology for distributed applications and systems is acceptable scalability at a low cost (no central servers are needed) and a potential for privacy preserving solutions, where data never leaves the computer of a user in a raw form. The label P2P covers a wide range of distributed algorithms that follow a specific system model, in which there are only minimal assumptions about the reliability of communication and the network components. A typical P2P system consists of a very large number of nodes (peers) that communicate via message passing. Messages can be delayed or lost, and peers can join and leave the system at any time.

In recent years, an intense effort has been made to develop collaborative machine learning algorithms that can be applied in P2P networks. This was motivated by the various potential applications such as spam filtering, user profile analysis, recommender systems and ranking. For example, for a P2P platform that offers rich functionality to its users including spam filtering, personalized search, and recommendation [6,15,39], or for P2P approaches for detecting distributed attack vectors [16], complex predictive models have to be built based on fully distributed, and often sensitive, data.

An important special case of P2P data processing is fully distributed data, where each node holds only one data record containing personal data, preferences, ratings, history, local sensor readings, and so on. Often, these personal data records are the most sensitive ones, so it is essential that we process them locally. At the same time, the learning algorithm has to be fully distributed, since the usual approach of building local models and combining them is not applicable.

Our goal here is to present algorithms for the case of fully distributed data. The design requirements specific to the P2P aspect are the following. First, the algorithm has to be extremely *robust*. Even in extreme failure scenarios it should maintain a reasonable performance. Second, prediction should be possible at any time in a *local* manner; that is, all nodes should be able to perform high quality prediction immediately without any extra communication. Third, the algorithm has to have a *low communication complexity*; both in terms of the number of messages sent, and the size of these messages as well. Privacy preservation is also one of our main goals, although in this study we do not analyze this aspect explicitly.

The gossip learning approach we propose involves models that perform a random walk in the P2P network, and that are updated each time they visit a node, using the local data record. 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. Since models perform random walks, all nodes will experience a continuous stream of models passing through them. Apart from using these models for prediction directly, nodes can also combine them in various ways using ensemble learning.

The generic skeleton of gossip learning involves three main components, namely an implementation of random walk, an online learning algorithm, and ensemble learning. Here we focus on an instantiation of gossip learning, where the online learning method is a stochastic gradient descent for linear models. In addition, nodes do not simply update and then pass on models during the random walk, but they also combine these models in the process. This implements a distributed "virtual" ensemble learning method similar to bagging, in which we in effect calculate a weighted voting over an exponentially increasing number of linear models.

Our specific contributions include the following: (1) we propose gossip learning, a novel and generic approach for P2P learning on fully distributed data, which can be instantiated in various ways; (2) we introduce a novel, efficient distributed ensemble learning method for linear models that virtually combines an exponentially increasing number of linear models; and (3) we provide a theoretical and empirical analysis of the convergence properties of the method in various scenarios.

The outline of this chapter is as follows. Section 2.2 elaborates on the fully distributed data model. Section 2.3 summarizes related work and the background concepts. In Section 2.4 we describe our generic approach and give a naive algorithm as an example. Section 2.5 presents the core algorithmic contributions of this chapter along with a theoretical discussion, followed by an experimental analysis in Section 2.6. And Section 2.7 concludes the chapter.

2.2 Fully Distributed Data

Our focus is on fully distributed data, where each node in the network has a single feature vector that cannot be moved to a server or to other nodes. Since this model is unusual in the data mining community, we elaborate on the motivation and the implications of the model.

In the distributed computing literature the fully distributed data model is typical. In the past decade, several algorithms have been proposed to calculate distributed aggregation queries over fully distributed data, such as the average, the maximum, and the network size (e.g., [12, 26, 49]). Here, the motivation for not moving any raw data is mainly to achieve robustness and adaptivity through not relying on any central servers. In some systems, like sensor networks or mobile ad hoc networks, the physical constraints on communication also prevent the collection of the data.

An additional motivation for not moving data is *privacy preservation*, where local data is not revealed in its raw form, even if the computing infrastructure made it possible. This is especially important in P2P social networking [19], where the key motivation is to give the user full control over personal data. Clearly, in P2P social networks, there is a need for more complex aggregation queries, and ultimately, for data models, to support features such as recommendations and spam filtering, and to make the system more robust with the help of, for example, distributed intruder detection. In other fully distributed systems data models are also important for monitoring and control.

Motivated by the emerging need for building complex data models over fully distributed data in different systems, we work with the abstraction of fully distributed data, and we aim at proposing generic algorithms that are applicable in all compatible systems.

In the fully distributed model, the requirements of an algorithm also differ from those of parallel data mining algorithms, and even from previous work on P2P data mining. Here, the decisive factor is the cost of message passing. Besides, the number of messages each node is allowed to send in a given time window is limited, so computation that is performed locally has a cost that is typically negligible when compared to communication delays. For this reason prediction performance has to be investigated *as a function of the number of messages sent*, as opposed to wall clock time. Since communication is crucially important, evaluating robustness to communication failures (such as message delay and message loss) also gets a large emphasis.

The approach we present here can also be appplied when each node stores many records (i.e. not just one); but its advantages to known approaches to P2P data mining become less significant, since communication plays a smaller role when local data is already usable to build reasonably good models. In the following we focus on the fully distributed model.

2.3 Background and Related Work

We organize the discussion of the background of our work along the generic model components outlined in the Introduction and explained in Section 2.4: online learning, ensemble learning and peer sampling. We also discuss related work in P2P data mining. Here, we do not consider parallel data mining algorithms. This field has a large literature, but the rather different underlying system model means it is of little relevance to us here.

Online Learning. Gossip learning relies on algorithms that iterate over the available training data, or process a continuous stream of data records, and evolve a model by updating it for each individual data record according to some update rule. Ma et al. provide a nice summary of such algorithms for large scale data [32]. Among these methods, we focus on stochastic gradient descent, which is a simple algorithm that has been shown to be very well suited for the large scale data mining scenarios we are interested in [10,11]. In this chapter, we use the Pegasos algorithm, which is a gradient method for linear SVM learning [42].

Ensemble Learning. Most distributed large scale algorithms apply some form of ensemble learning to combine models learned over different samples of the training data. Rokach presents a survey of ensemble learning methods [41]. We apply a method for combining the models in the network that is related to both bagging [13] and "pasting small votes" [14]: when the models start their random walk, initially they are based on non-overlapping small subsets of the training data due to the large scale of the system (the key idea behind pasting small votes) and as time goes by, the sample sets grow, approaching the case of bagging (although the samples that belong to different models will not be completely independent in our case).

Peer Sampling in Distributed Systems. The sampling probability for each data record is defined by peer sampling algorithms that are used to implement the random walk. Here, we apply uniform sampling. A set of approaches to implement uniform sampling in a P2P network apply random walks themselves over a fixed overlay network, in such a way that the corresponding Markov-chain has a uniform limiting distribution [22, 28, 46]. In our algorithm, we apply gossip-based peer sampling [27] where peers periodically exchange small random subsets of addresses, thereby providing a local random sample of the addresses at each point in time at each node. The advantage of gossip-based sampling in our setting is that samples are available locally and without delay. Furthermore, the messages related to the peer sampling algorithm can piggyback the random walks of the models, thereby avoiding any overheads in terms of message complexity.

P2P Learning. 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: Astrolabe [49] and gossip-based averaging [26]. 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 [30], unsupervised learners [44] or the collaborative filtering based recommender algorithms [7,23,36,48]. However, here we seek to provide a rather generic approach that covers a wide range of machine learning models, while maintaining a similar robustness and simplicity.

In the past few years there has been an increasing number of proposals for P2P machine learning algorithms as well, like those in [3–5, 18, 24, 31, 45]. The usual assumption in these studies is that a peer has a subset of the training data on which a

Algorithm 1 Gossip Learning Scheme

```
1: initModel()
2: loop
3:     wait($\Delta$)
4:     $p \leftarrow$ selectPeer()
5:     send modelCache.freshest() to $p$
6: end loop
7:
8: procedure ONRECEIVEMODEL($m$)
9:     modelCache.add(createModel($m$, lastModel))
10:    lastModel $\leftarrow m$
11: end procedure
```

model can be learned locally. After learning the local models, algorithms either aggregate the models to allow each peer to perform local prediction, 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 changing data. In our approach we adopt the fully distributed model, where each node holds only one data record. In this case we cannot talk about local learning: every aspect of the learning algorithm is inherently distributed. Since we assume that data cannot be moved, the models need to visit data instead. In a setting like this, the main problem we need to solve is to efficiently aggregate the various models that evolve slowly in the system so as to speed up the convergence of prediction performance.

To the best of our knowledge there is no other learning approach available that is designed to work in our fully asynchronous and unreliable message passing model, and which is capable of producing a large array of state-of-the-art models.

2.4 Gossip Learning: the Basic Idea

Algorithm 1 provides the skeleton of the gossip learning framework. The same algorithm is run at each node in the network. The algorithm consists of an active loop of periodic activity, and a method for handling incoming models. Based on every incoming model a new model is created potentially combining it with the previous incoming model. This newly created model is stored in a cache of a fixed size. When the cache is full, the model stored for the longest time is replaced by the latest model added. The cache provides a pool of recent models that can be used to implement, for example, voting based prediction. We discuss this possibility in Section 2.6. In the active loop the freshest model (the model added to the cache most recently) is sent to a random peer.

We make no assumptions about either the synchrony of the loops at the different nodes or the reliability of the messages. We do assume that the period of the loop Δ is the same at every node. For simplicity, here we assume that the active loop is initiated at the same time at every node, and we do not consider any stopping criteria, so the loop runs indefinitely. The assumption about the synchronized start allows us to focus on the convergence properties of the algorithm, but it is not a crucial requirement in

Algorithm 2 CREATEMODEL: three implementations

```
1:  procedure CREATEMODELRW($m_1, m_2$)
2:      return update($m_1$)
3:  end procedure
4:
5:  procedure CREATEMODELMU($m_1, m_2$)
6:      return update(merge($m_1, m_2$))
7:  end procedure
8:
9:  procedure CREATEMODELUM($m_1, m_2$)
10:     return merge(update($m_1$),update($m_2$))
11: end procedure
```

practical applications. In fact, randomly restarted loops actually help in following drifting concepts and changing data, which is the subject of our ongoing work.

The algorithm contains abstract methods that can be implemented in different ways to obtain a concrete learning algorithm. The main placeholders are SELECTPEER and CREATEMODEL. Method SELECTPEER is the interface for the peer sampling service, as described in Section 2.3. Here we use the NEWSCAST algorithm [27], which is a gossip-based implementation of peer sampling. We do not discuss NEWSCAST here in detail, all we assume is that SELECTPEER() provides a *uniform random sample* of the peers without creating *any extra messages* in the network, given that NEWSCAST gossip messages (which contain only a few dozen network addresses) can piggyback gossip learning messages.

The core of the approach is CREATEMODEL. Its task is to create a new updated model based on locally available information—the two models received most recently, and the local single training data record—to be sent on to a random peer. Algorithm 2 lists three implementations that are still abstract. They represent those three possible ways of breaking down the task that we will study in this deliverable.

The abstract method UPDATE represents the online learning algorithm—the second main component of our framework besides peer sampling—that updates the model based on one example (the local example of the node). Procedure CREATEMODELRW implements the case where models independently perform random walks over the network. We will use this algorithm as a baseline.

The remaining two variants apply a method called MERGE, either before the update (MU) or after it (UM). Method MERGE helps implement the third component, namely ensemble learning. A *completely impractical* example for an implementation of MERGE is the case where the model space consists of all the sets of basic models of a certain type. Then MERGE can simply merge the two input sets, UPDATE can update all the models in the set, and a prediction can be made by, for example, majority voting (for classification) or averaging the predictions (for regression). With this implementation, all nodes would collect an exponentially increasing set of models, allowing for a much better prediction after a much shorter learning time in general that based on a single model [13, 14], although the learning history for the members of the set would not be completely independent.

This implementation is, of course, impractical because the size of the messages in each cycle of the main loop would increase exponentially. Our main contribution is to discuss and analyze a special case, namely linear models. For linear models we will propose an algorithm where the message size can be kept *constant*, while producing the same (or similar) behavior as the impractical implementation above. The subtle difference between the MU and UM versions will also be discussed.

Let us close this section with a brief analysis of the cost of the algorithm in terms of computation and communication. As of communication: each node in the network sends exactly one message in each Δ time units. The size of the message depends on the selected hypothesis space; normally it contains the parameters of a single model. In addition, the message also contains a small, constant number of network addresses as defined by the NEWSCAST protocol (typically around 20). The computational cost is one or two update steps in each Δ time units for the UM or the MU variants, respectively. The exact cost of this step depends on the selected online learner.

2.5 Merging Linear Models through Averaging

The key observation we make is that in a linear hypothesis space, in certain cases voting-based prediction is equivalent to a single prediction by taking the *average* of the models that participate in the voting. Furthermore, updating a set of linear models and then averaging them is sometimes equivalent to averaging the models first, and then updating the resulting single model. These observations are valid in a strict sense only in special circumstances. However, our intuition is that even if this key observation holds only in a heuristic sense, it still provides a valid heuristic explanation of the behavior of the resulting averaging-based merging approach.

In the following we first give an example of a case where there is a strict equivalence of averaging and voting to illustrate the concept, then we discuss and analyze a practical and competitive algorithm, where the correspondence of voting and averaging is only heuristic in nature.

2.5.1 The Adaline Perceptron

Here, we consider the Adaline perceptron [51] which arguably has one of the simplest update rules due to its linear activation function. Without loss of generality, we ignore the bias term. The error function to be optimized is defined as

$$E_x(w) = \frac{1}{2}(y - \langle w, x\rangle)^2, \tag{2.1}$$

where w is the linear model and (x, y) is a training example ($x, w \in \mathbb{R}^n$, $y \in \{-1, 1\}$). The gradient at w for x is given by

$$\nabla_w = \frac{\partial E_x(w)}{\partial w} = -(y - \langle w, x\rangle)x \tag{2.2}$$

that defines the learning rule for (x, y) by

$$w^{(k+1)} = w^{(k)} + \eta(y - \langle w^{(k)}, x\rangle)x, \tag{2.3}$$

where η is the learning rate. In this case it is a constant.

Now, let us assume that we are given a set of models $w_1, \ldots, w_m$, and let us define $\bar{w} = (w_1 + \ldots + w_m)/m$. In the case of a regression problem, the prediction for a given point x and model w is $\langle w, x\rangle$. It is not hard to see that

$$h(x) = \langle \bar{w}, x\rangle = \frac{1}{m}\langle \sum_{i=0}^{m} w_i, x\rangle = \frac{1}{m}\sum_{i=0}^{m}\langle w_i, x\rangle, \tag{2.4}$$

which means that the voting-based prediction is equivalent to prediction based on the average model.

In the case of classification, the equivalence does not hold for all voting mechanisms. But it is easy to verify that in the case of a weighted voting approach, where vote weights are given by $|\langle w, x\rangle|$, and the votes themselves are given by $\operatorname{sgn}\langle w, x\rangle$, the same equivalence holds:

$$h(x) = \operatorname{sgn}(\frac{1}{m}\sum_{i=1}^{m}|\langle w, x\rangle| \operatorname{sgn}\langle w, x\rangle) = \operatorname{sgn}(\frac{1}{m}\sum_{i=1}^{m}\langle w_i, x\rangle) = \operatorname{sgn}\langle \bar{w}, x\rangle. \tag{2.5}$$

A similar approach to this weighted voting mechanism has been shown to improve the performance of simple vote counting [8]. Our preliminary experiments also support this.

In a very similar manner, it can be shown that updating $\bar{w}$ using an example (x, y) is equivalent to updating all the individual models $w_1, \ldots, w_m$ and then taking the average:

$$\bar{w} + \eta(y - \langle \bar{w}, x\rangle)x = \frac{1}{m}\sum_{i=1}^{m} w_i + \eta(y - \langle w_i, x\rangle)x. \tag{2.6}$$

The above properties lead to a rather important observation. If we implement our gossip learning skeleton using Adaline, as shown in Algorithm 3, then the resulting algorithm behaves exactly as if all the models were simply stored and then forwarded, resulting in an exponentially increasing number of models contained in each message, as described in Section 2.4. That is, averaging effectively reduces the exponential message complexity to transmitting a *single* model in each cycle independently of time, yet we enjoy the benefits of the aggressive, but impractical approach of simply replicating all the models and using voting over them for prediction.

It should be mentioned that—even though the number of "virtual" models grows exponentially fast—the algorithm is not equivalent to bagging over an exponential number of independent models. In each gossip cycle, there are only N independent updates occurring in the system overall (where N is the number of nodes), and the effect of these updates is being aggregated rather efficiently. In fact, as we will see in Section 2.6, bagging over N independent models actually outperforms the gossip learning algorithms.

2.5.2 Pegasos

Here, we discuss the adaptation of Pegasos (a linear SVM gradient method [42] used for classification) into our gossip framework. The components required for the adaptation are shown in Algorithm 3, where method UPDATEPEGASOS is simply taken from [42]. For a complete implementation of the framework, one also needs to select an implementation of CREATEMODEL from Algorithm 2. In the following, the three

Algorithm 3 Pegasos and Adaline updates, initialization, and merging

```
1: procedure UPDATEPEGASOS($m$)
2:     $m.t \leftarrow m.t + 1$
3:     $\eta \leftarrow 1/(\lambda \cdot m.t)$
4:     if $y \langle m.w, x \rangle < 1$ then
5:         $m.w \leftarrow (1 - \eta\lambda)m.w + \eta y x$
6:     else
7:         $m.w \leftarrow (1 - \eta\lambda)m.w$
8:     end if
9:     return $m$
10: end procedure
11:
12: procedure UPDATEADALINE($m$)
13:     $m.w \leftarrow m.w + \eta(y - \langle m.w, x \rangle)x$
14:     return $m$
15: end procedure
16:
17: procedure INITMODEL
18:     lastModel.$t \leftarrow 0$
19:     lastModel.$w \leftarrow (0, \dots, 0)^T$
20:     modelCache.add(lastModel)
21: end procedure
22:
23: procedure MERGE($m_1$,$m_2$)
24:     $m.t \leftarrow \max(m_1.t, m_2.t)$
25:     $m.w \leftarrow (m_1.w + m_2.w)/2$
26:     return $m$
27: end procedure
```

versions of a complete Pegasos-based implementation defined by these options will be referred to as P2PEGASOSRW, P2PEGASOSMU, and P2PEGASOSUM.

The main difference between the Adaline perceptron and Pegasos is the context dependent update rule that is different for correctly and incorrectly classified examples. Due to this difference, there is no strict equivalence between averaging and voting, as in the previous subsection. To see this, consider two models, w_1 and w_2, and an example (x, y), and let $\bar{w} = (w_1 + w_2)/2$. In this case, updating w_1 and w_2 first, and then averaging them results in the same model as updating $\bar{w}$ if and only if both w_1 and w_2 classify x in the same way (correctly or incorrectly). This is because when updating $\bar{w}$, we virtually update both w_1 and w_2 in the same way, irrespective of how they classify x individually.

This seems to suggest that P2PEGASOSUM is a better choice. We will test this hypothesis experimentally in Section 2.6, where we will show that, surprisingly, it is not always true. The reason could be that P2PEGASOSMU and P2PEGASOSUM are in fact very similar when we consider the entire history of the distributed computation, as opposed to a single update step. The histories of the models define a directed acyclic graph (DAG), where the nodes are merging operations, and the edges correspond to the transfer of a model from one node to another. In both cases, there is one update corresponding to each edge: the only difference is whether the update occurs on the source node of the edge or on the target. Apart from this, the edges of the DAG are the same for both methods. Hence we see that P2PEGASOSMU has the favorable property that the updates that correspond to the incoming edges of a merge operation are done using independent samples, while for P2PEGASOSUM they are performed with the same example. Thus, P2PEGASOSMU guarantees a greater independence of the models.

In the following we present our theoretical results for both P2PEGASOSMU and P2PEGASOSUM. We note that these results do not assume any coordination or synchronization; they are based on a fully asynchronous communication model. First let us formally define the optimization problem, and let us introduce some notation.

Let $S = \{(x_i, y_i) : 1 \leq i \leq m, x_i \in \mathbb{R}^n, y_i \in \{+1, -1\}\}$ be a distributed training set with one data point at each network node. Let $f : \mathbb{R}^n \to \mathbb{R}$ be the objective function of the SVM learning problem:

$$\begin{gathered} f(w) = \min_w \frac{\lambda}{2}\|w\|^2 + \frac{1}{m}\sum_{(x,y)\in\mathcal{S}} \ell(w;(x,y)), \\ \text{where } \ell(w;(x,y)) = \max\{0, 1 - y\langle w, x\rangle\} \end{gathered} \tag{2.7}$$

Note that f is strongly convex with a parameter λ [42]. Let $w^\star$ denote the global optimum of f. For a fixed data point (x_i, y_i) we define

$$f_i(w) = \frac{\lambda}{2}\|w\|^2 + \ell(w;(x_i, y_i)), \tag{2.8}$$

which is used to derive the update rule for the Pegasos algorithm. Obviously, f_i is λ strongly convex as well, since it has the same form as f with $m = 1$.

The update history of a model can be represented as a binary tree, where the nodes are models and the edges are defined by the direct ancestor relation. Let us denote the direct ancestors of $w^{(i+1)}$ by $w_1^{(i)}$ and $w_2^{(i)}$. These ancestors are averaged and then updated to obtain $w^{(i+1)}$ (assuming the MU variant). Let the sequence $w^{(0)}, \ldots, w^{(t)}$ be defined as the path in this history tree, for which

$$w^{(i)} = \text{argmax}_{w\in\{w_1^{(i)}, w_2^{(i)}\}}\|w - w^\star\|, \quad i = 0, \ldots, t-1. \tag{2.9}$$

This sequence is well defined. Let (x_i, y_i) denote the training example that was used in the update step, and which gave $w^{(i)}$ in the series defined above.

Theorem 1 (P2PEGASOSMU convergence)**.** *We assume that (1) each node receives an incoming message after any point in time within a finite time period (eventual update assumption), (2) there is a subgradient ∇ of the objective function such that $\|\nabla_w\| \leq G$ for every w. Then,*

$$\frac{1}{t}\sum_{i=1}^{t} f_i(\bar{w}^{(i)}) - f_i(w^\star) \leq \frac{G^2(log(t)+1)}{2\lambda t} \tag{2.10}$$

where $\bar{w}^{(i)} = (w_1^{(i)} + w_2^{(i)})/2$.

Proof. During the running of the algorithm, let us pick any node on which at least one subgradient update has already been performed. There is such a node eventually, due to the eventual update assumption. Let the model currently stored at this node be $w^{(t+1)}$.

We know that $w^{(t+1)} = \bar{w}^{(t)} - \nabla^{(t)}/(\lambda t)$, where $\bar{w}^{(t)} = (w_1^{(t)} + w_2^{(t)})/2$ and where $\nabla^{(t)}$ is the subgradient of f_t. From the λ-convexity of f_t it follows that

$$f_t(\bar{w}^{(t)}) - f_t(w^\star) + \frac{\lambda}{2}\|\bar{w}^{(t)} - w^\star\|^2 \leq \langle \bar{w}^{(t)} - w^\star, \nabla^{(t)} \rangle. \tag{2.11}$$

On the other hand, the following inequality is also true, following from the definition of $\bar{w}^{(t+1)}$, G and some algebraic rearrangement:

$$\langle \bar{w}^{(t)} - w^\star, \nabla^{(t)} \rangle \leq \frac{\lambda t}{2}\|\bar{w}^{(t)} - w^\star\|^2 - \frac{\lambda t}{2}\|w^{(t+1)} - w^\star\|^2 + \frac{G^2}{2\lambda t}. \tag{2.12}$$

Moreover, we can bound the distance of $\bar{w}^{(t)}$ from $w^\star$ by the distance of the ancestor of $\bar{w}^{(t)}$ that is further away from $w^\star$ with the help of the Cauchy–Bunyakovsky–Schwarz inequality:

$$\|\bar{w}^{(t)} - w^\star\|^2 = \left\| \frac{w_1^{(t)} - w^\star}{2} + \frac{w_2^{(t)} - w^\star}{2} \right\|^2 \leq \|w^{(t)} - w^\star\|^2. \tag{2.13}$$

From (2.11), (2.12), (2.13) and the bound on the subgradients, we have

$$f_t(\bar{w}^{(t)}) - f_t(w^\star) \leq \frac{\lambda(t-1)}{2}\|w^{(t)} - w^\star\|^2 - \frac{\lambda t}{2}\|w^{(t+1)} - w^\star\|^2 + \frac{G^2}{2\lambda t}. \tag{2.14}$$

Note that this bound also holds for $w^{(i)}$, $1 \leq i \leq t$. Summing up both sides of these t inequalities, we get the following bound:

$$\sum_{i=1}^{t} f_i(\bar{w}^{(i)}) - f_i(w^\star) \leq -\frac{\lambda t}{2}\|w^{(t+1)} - w^\star\|^2 + \frac{G^2}{2\lambda}\sum_{i=1}^{t}\frac{1}{i} \leq \frac{G^2(log(t)+1)}{2\lambda}, \tag{2.15}$$

from which the theorem follows after division by t. $\square$

The bound in (2.15) is analogous to the bound presented in [42] in the analysis of the PEGASOS algorithm. It basically means that the average error tends to zero. To be able to show that the limit of the process is the optimum of f, it is necessary that the samples involved in the series are uniform random samples [42]. Investigating the distribution of the samples is left to future work; but we believe that the distribution closely approximates uniformity for a large t, given the uniform random peer sampling that is applied.

For P2PEGASOSUM, an almost identical derivation leads us to a similar result (omitted due to lack of space).

2.6 Experimental Results

We experiment with two algorithms, namely P2PEGASOSUM and P2PEGASOSMU. In addition, to shed light on the behavior of these algorithms, we include a number of baseline methods as well. In the experiments that we performed, we used the PEERSIM event based P2P simulator [38].

2.6.1 Experimental Setup

Baseline Algorithms. The first baseline we use is P2PEGASOSRW. If there is no message drop or message delay, this is equivalent to the Pegasos algorithm, since in cycle t all peers will have models that are the result of Pegasos learning on t random examples. In the case of message delay and message drop failures, the number of samples will be less than t, as a function of the drop probability and the delay.

We also examine two variants of *weighted bagging*. The first variant (WB1) is defined as

$$h_{WB1}(x,t) = \mathrm{sgn}(\sum_{i=1}^{N} \langle x, w_i^{(t)} \rangle), \tag{2.16}$$

where N is the number of nodes in the network, and the linear models $w_i^{(t)}$ are learned with Pegasos over an independent sample of size t of the training data. This baseline algorithm can be thought of as the ideal utilization of the N independent updates performed in parallel by the N nodes in the network in each cycle. The gossip framework introduces dependencies among the models, so its performance can be expected to be worse.

In addition, in the gossip framework a node has influence from only 2^t models on average in cycle t. To account for this handicap, we also use a second version of weighted bagging (WB2):

$$h_{WB2}(x) = \mathrm{sgn}(\sum_{i=1}^{\min(2^t,N)} \langle x, w_i \rangle). \tag{2.17}$$

The weighted bagging variants described above are not practical alternatives, these algorithms serve as a baseline only. The reason is that an actual implementation would require N independent models for prediction. This could be achieved by P2PEGASOSRW with a distributed prediction, which would impose a large cost and delay for each prediction. This could also be achieved by all nodes running up to $O(N)$ instances of P2PEGASOSRW, and using the $O(N)$ local models for prediction; this is not feasible either. In sum, the point that we want to make is that our gossip algorithm approximates WB2 quite well using only a single message per node in each cycle, due to the technique of merging models.

The last baseline algorithm we experiment with is PERFECT MATCHING. In this algorithm we replace the peer sampling component of the gossip framework: instead of each node picking random neighbors for each cycle, we create a random perfect matching among the peers so that every peer receives exactly one message. Our hypothesis was that—since this variant increases the efficiency of mixing—it will maintain a higher diversity of models, and so a better performance can be expected due to the "virtual bagging" effect we explained previously. Note that this algorithm is not intended for practical use either.

Data Sets. We used three different data setsn, namely Reuters [21], Spambase, and the Malicious URLs [32] data sets, which were obtained from the UCI database repository [20]. These data sets are all quite different, and include small and large sets containing a small or large number of features. Table 2.1 shows the main properties of these data sets, as well as the prediction performance of the Pegasos algorithm.

Table 2.1: The main properties of the data sets, and the prediction error of the baseline sequential algorithm.

	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)

Algorithm 4 Local prediction procedures

```
1:  procedure PREDICT($x$)
2:      $w \leftarrow$ modelCache.freshest()
3:      return sign($\langle w, x \rangle$)
4:  end procedure
5:
6:  procedure VOTEDPREDICT($x$)
7:      pRatio $\leftarrow 0$
8:      for $m \in$ modelCache do
9:          if sign($\langle m.w, x \rangle$) $\geq 0$ then
10:             pRatio $\leftarrow$ pRatio $+1$
11:         end if
12:     end for
13:     return sign(pRatio/modelCache.size()$-0.5$)
14: end procedure
```

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 [26], but we performed it offline. The effect of this dramatic reduction on the prediction performance is shown in Table 2.1, where Pegasos results on the full feature set are shown in parenthesis.

Using the local models for prediction. An important aspect of our protocol is that each node has at least one model available locally, and thus all the nodes can perform a prediction. Moreover, since the nodes can remember the models that pass through them at no communication cost, we cheaply implement a simple voting mechanism, where nodes will use more than one model to make predictions. Algorithm 4 shows the procedures used for prediction in the original case, and in the case of voting. Here the vector x is the unseen example to be classified. In the case of linear models, the classification is simply the sign of the inner product with the model, which essentially describes on which side of the hyperplane the given point lies. In our experiments we used a cache size of 10.

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 ratios of this splitting are shown in Table 2.1. In our experiments with P2PEGASOSMU and P2PEGASOSUM 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.

For the baseline algorithms we used all the available models to calculate the error rate, which equals to the number of training samples. From the Malicious URLs database we used only 10,000 examples selected at random, to make the evaluation computationally feasible. Note, that we found that increasing the number of examples beyond 10,000 does not result in any noticeable difference in the observed behavior.

We also calculated the similarities between the models circulating in the network, using the cosine similarity measure. We calculated the similarity between all pairs of models, and calculated the average. This metric is useful to study the speed at which the actual models converge. Note that under uniform sampling it is known that all models converge to an optimal model.

Modeling failure. In our 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 $[\Delta, 10\Delta]$, where Δ is the gossip period in Algorithm 1. 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.

2.6.2 Results and Discussion

The experimental results for prediction without local voting are shown in Figures 2.1 and 2.2. Note that all variants can be mathematically proven to converge to the same result, so the difference is in convergence speed only. Bagging can temporarily outperform a single instance of Pegasos, but after enough training samples, all models become almost identical, so the advantage of voting disappears.

In Figure 2.1 we can see that our hypothesis about the relationship between the performance of the gossip algorithms and the baselines is validated: the standalone Pegasos algorithm is the slowest, while the two variants of weighted bagging are the fastest. P2PEGASOSMU approximates WB2 quite well, with some delay, so we can use WB2 as a heuristic model of the behavior of the algorithm. Note that the convergence is several orders of magnitude faster than that of Pegasos (the plots have a logarithmic scale).

Figure 2.1 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 2.2 illustrates the difference between the UM and MU variants. Here we model no failures. In Section 2.5.2 we pointed out that—although the UM version

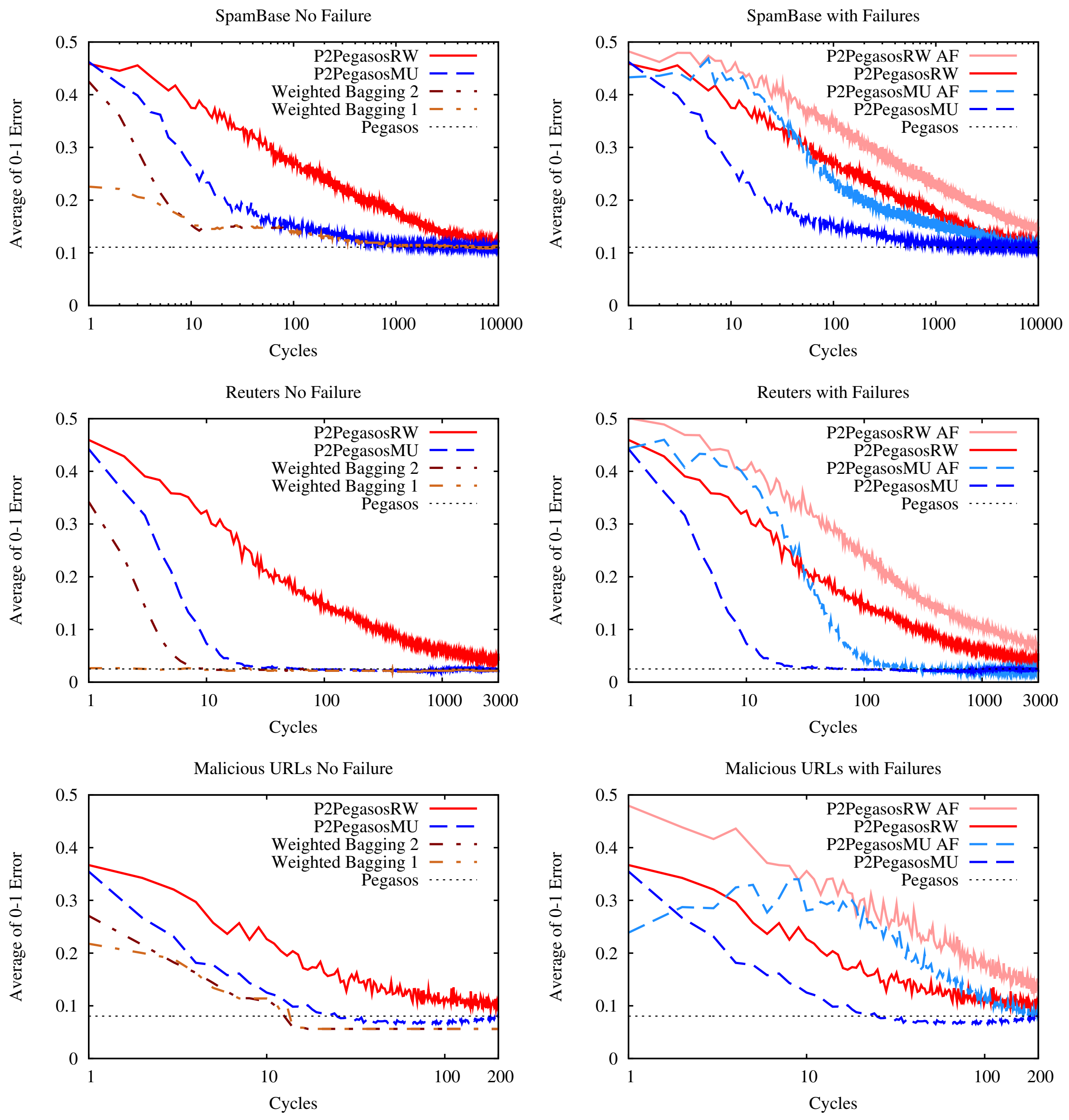

Figure 2.1: Experimental results without failure (left hand side) and with extreme failure (right hand side). AF means all possible failures are modeled.

looks favorable when considering a single node—when looking at the full history of the learning process P2PEGASOSMU maintains more independence between the models. Indeed, the MU version clearly performs better according to our experiments. We can also observe that the UM version shows a lower level of model similarity in the system, which probably has to do with the slower convergence.

In Figure 2.2 we can see the performance of the perfect matching variant of P2PEGASOSMU as well. Contrary to our expectations, perfect matching does not clearly improve performance, apart from the first few cycles. It is also interesting to observe, that model similarity is correlated to prediction performance also in this case. We also note, that in the case of the Adaline-based gossip learning implementa-

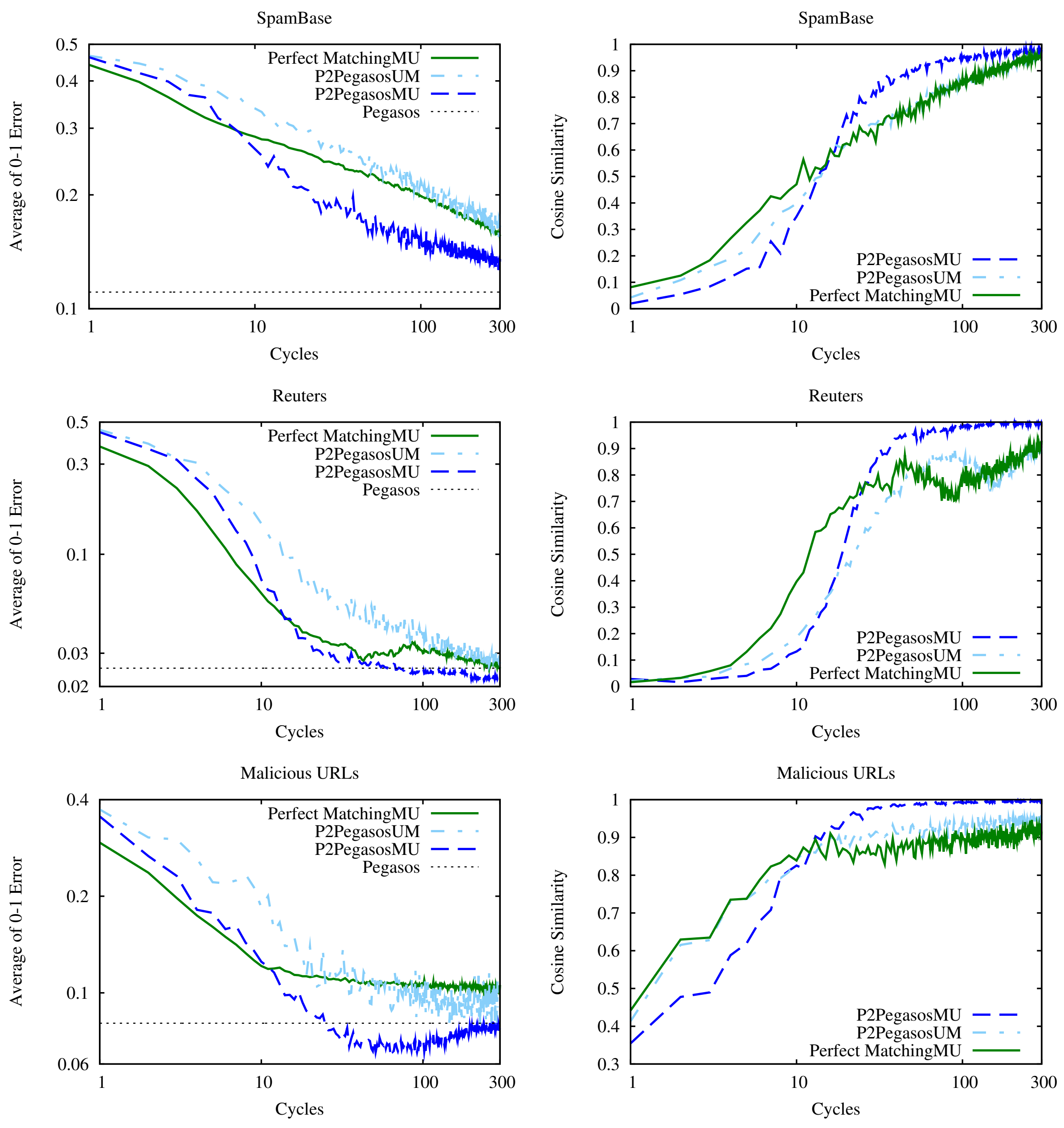

Figure 2.2: Prediction error (left hand side) and model similarity (right hand side) with PERFECT MATCHING and P2PEGASOSUM.

tion perfect matching is clearly better than random peer sampling (not shown). This means that this behavior is due to the context-dependence of the update rule discussed in 2.5.2.

The results with local voting are shown in Figure 2.3. The main conclusion is that voting results in a significant improvement when applied along with P2PEGASOSRW, the learning algorithm that does not apply merging. When merging is applied, the improvement is less dramatic. In the first few cycles, voting can result in a slight degradation of performance. This could be expected, since the models in the local caches are trained on fewer samples on average than the freshest model in the cache. Overall, since voting is for free, it is advisable to use it.

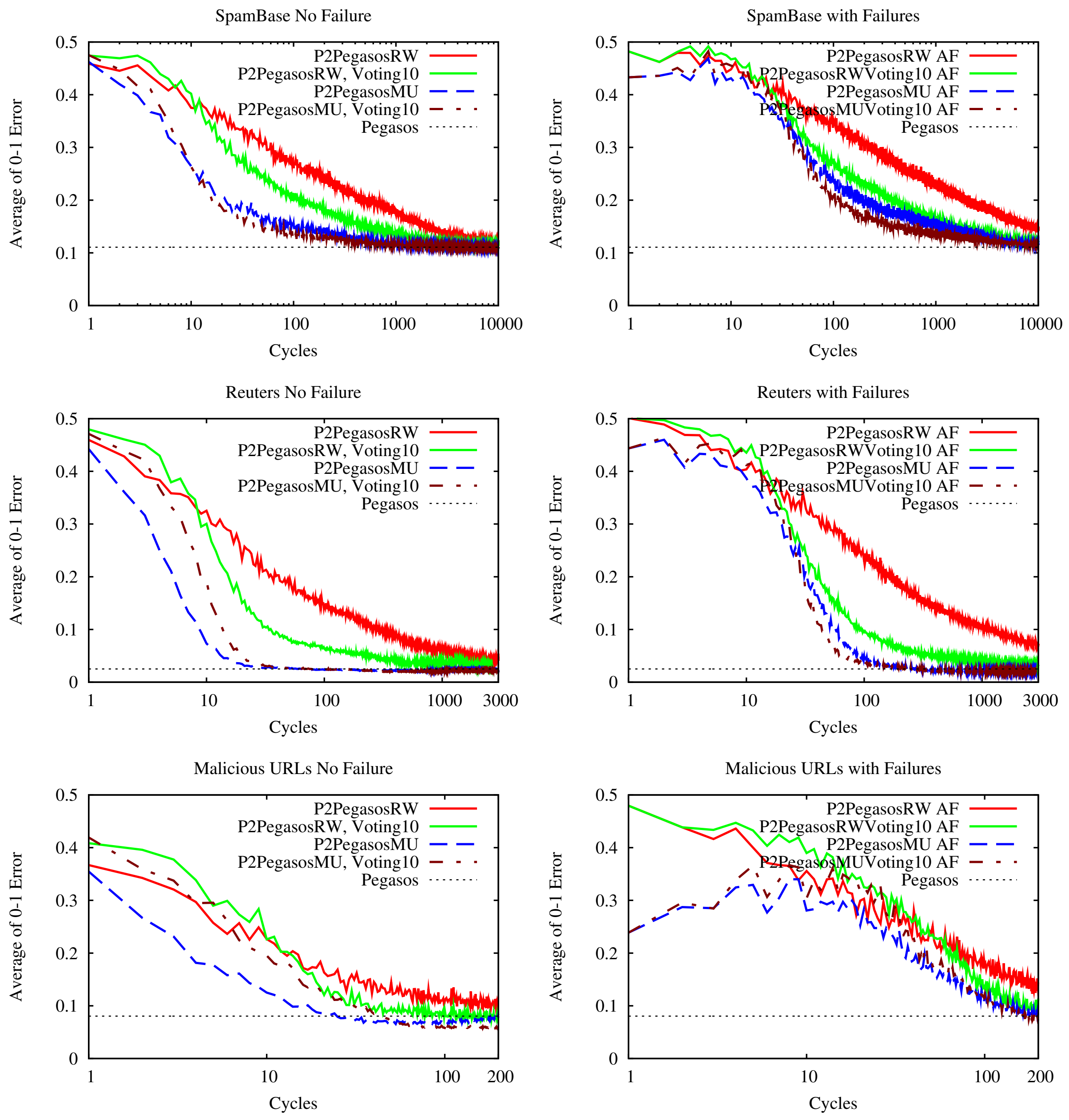

Figure 2.3: Experimental results of applying local voting without failure (left hand side) and with extreme failure (right hand side).

2.7 Conclusions

We proposed gossip learning as a generic approach for learning models of fully distributed data in large scale P2P systems. The basic idea of gossip learning is that many models perform a random walk over the network, while being updated at each node they visit, and while being combined (merged) with other models they encounter. We presented an instantiation of gossip learning based on the Pegasos algorithm. The algorithm was shown to be extremely robust to message drop and message delay, furthermore, a very significant speedup was demonstrated w.r.t. the baseline Pegasos algorithm due to the model merging technique and the prediction algorithm that is based on local voting.

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

Chapter 3

Implementation of the Gossip Learning Framework in QMedia

In collaboration with the Technical University of Delft, we have implemented the above-mentioned Gossip Learning Framework within the Tribler Peer-to-Peer client, which is the main platform of QMedia [50]. We provide an initial, proof-of-concept implementation that is modular and easily extendable.

This implementation covers the core implementation and technical details of Gossip Learning Framework instead of focusing on a certain learning scenario. It can be considered as an abstract layer between Tribler and the real learning protocols, which provides a clear interface for these upper protocols.

This implementation follows some of the conventions of the Gossip Learning Framework, available at

`http://github.com/RobertOrmandi/Gossip-Learning-Framework`

3.1 Motivation

Tribler is a social-based Peer-to-Peer system, BitTorrent client. When using this software, a lot of data is accumulated. We can learn on this data, by using gossip learning techniques, and improve the user experience by building intelligent spam filtering or personalized recommender systems. In this section we give a brief overview—-as a motivation—of what kind of end user applications can be built on the top of our implementation.

As we said, one of the most promising applications of the Gossip Learning Framework and its instantiation in Tribler is learning personalized recommendations based on the large amount of data distributedly available in networks. Here, we briefly describe the commonlyto applied algorithm for building recommender systems called Collaborative Filtering (CF) [1] and a possible initialization in the Tribler platform on the top of the Gossip Learning Framework subsystem. We would like to highlight that the followings have not yet been implemented, so the remaining part of this subsection can be viewed as an example of a potential future application.

3.1.1 Collaborative Filtering Overview

Several approaches exist for building recommender systems [1]. The diversity of the applied techniques is huge. Some approaches apply similarity based computation [36, 40], low rank matrix factorization method [25, 29, 47], clustering [37], probability models [17], and other machine learning techniques [37] or graph based methods [43, 52]. When we try to adapt these approaches to a distributed environment like the Tribler platform, we have to be careful since most of these approaches are too centralized to implement them in a distributed manner. For this reason, several relevant approaches cannot be implemented in Tribler [43, 52]. A promising technique is Collaborative Filtering, which has a gradient descent base solution, since it is a suitable candidate for adapting in the Gossip Learning Framework and Tribler.

In the problem statement of the Collaborative Filtering we assume that there is given a set of users U (e.g. users of the Tribler client) where $|U| = n$ and a set of items I (e.g. the torrent files in BitTorrent network like Tribler's one) where $|I| = m$. Moreover we assume that some of the users *rate* some of the items, which indicates how much the user likes the rated item. The rating value of user i on item j is denoted by $r_{i,j} \in \mathbb{R}$. The existing rates can be ordered into a matrix $R \in \mathbb{R}^{n \times m}$. Usually a small portion of the possible ratings is given in the system i.e. the number of defined values of R is much less than $n \times m$.

The aim of Collaborative Filtering is to fill the missing values of R automatically (i.e. without user interaction). We expect that these predicted rating values should be as close as possible to the real user preferences which are unknown in the system. It is easy to see that if we have good predicted user preferences on unseen items, we can simply recommend the top K unseen items to a user with the highest predicted rating values corresponding to the same user.

3.1.2 Implementation Idea

As mentioned above, Tribler is a modular and easily extendable P2P platform where a lot of data is accumulated. In the latest release of Tribler the users will be able to rate the torrents. This new feature makes Tribler a suitable platform for implementing a distributed recommender system. The ratings in the Tribler platform will be stored in a distributed manner (i.e. on the users' machines) which precludes the use of a centralized approache. Now, we propose a possible solution for building a distributed recommender system in Tribler based on the Gossip Learning Framework.

Our proposal is based on the low rank linear matrix factorization approach [25, 29, 47]. This techniques assumes that only a small number of features (the basis of the low ranked decomposition) influence the preferences. This idea can be formulated as an optimization problem which can bee seen in Eq. 3.1.

$$\min_{A \in \mathbb{R}^{n \times f}, B \in \mathbb{R}^{f \times m}} \frac{\lambda}{2}(\|A\|^2 + \|B\|^2) + \sum_{i,j: r_{i,j} \neq null} (r_{i,j} - \langle u_i, v_j \rangle)^2 \tag{3.1}$$

Here, the rating matrix R is approximated by the product of two lower dimensional matrices $A \in \mathbb{R}^{n \times f}$ and $B \in \mathbb{R}^{f \times m}$ which describe the user preferences and item characteristics, respectively, in an f-dimensional feature space where $f << \min(m, n)$.

Here λ is the regularization parameter, u_i is the ith row of matrix A and v_j is the jth column of matrix B.

The 3.1 problem can be solved by applying stochastic gradient descent method [25, 29,47], which can be fitted into the Gossip Learning Framework. This approach has the advantage that each user preference vector is stored on the user's machine who owns it and it is never moved. This is crucial in terms of privacy preservation. However, there are several open questions here. One is that the matrix B that describes the characteristics of the items is part of the model which is passed through the network. This could be a problem when the number of items is huge, since it increases the size of the model linearly. To keep this model size constant, we have to investigate this phenomenon more precisely. However, we think that applying a well-designed clustering on the items can solve the problem. These experiments will be in our focus in the near future.

In the followings we describe the actual, existing implementation details of Gossip learning Framework in Tribler.

3.2 Implementation details

Now, we provide a detailed discussion of the core of the Gossip Learning Framework, as implemented in Tribler as a *community*, and we also show two basic learning algorithms that work on top of the core. Before that, we introduce some basic concepts of the QLectives Platform.

3.2.1 Communities in Tribler

The QLectives Platform's version 2.0 introduces the Distributed Permission System (Dispersy) [50] as part of Tribler. Dispersy provides a platform to build up communities. A community is basically a protocol over a set of nodes. This includes the permission, distribution, and gossiping sub-systems. An example community is the Barter Community, introduced in [2]. Our contribution is the Gossip Learning Framework Community, which is described in detail in the next subsection.

3.2.2 General overview of the GossipLearningFramework community

We have used the 5.4.x version of Tribler, which was the stable release at the beginning of the collaboration. The later versions will contain some bugfixes and improvements. We follow the version changes and port the community over time.

We have implemented a new community called `GossipLearningCommunity`, which provides a generic framework for learning using gossiping. Each community must define a number of message types that it handles. In our case, there is only one message type called `modeldata`. We defined this message to contain one object of type `GossipMessage`, which is the generic base class of the learning models. The whole community uses this abstract message class, which can later be defined to be any learning algorithm.

Each community has to define message payload conversion, that is, the encoding and decoding of the message over the wire. We serialize our model objects using a

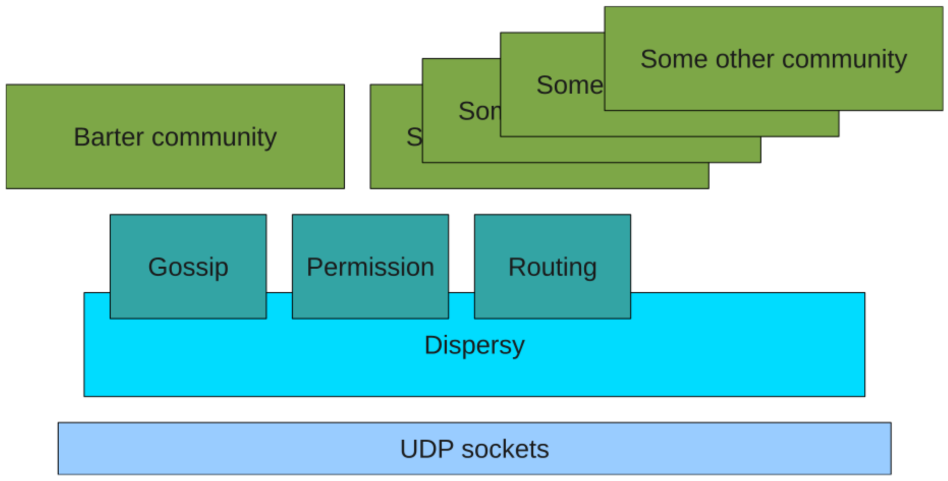

Figure 3.1: A schematic view of Dispersy and other components [50].

Listing 3.1: The modeldata message type definition.

```
Message(self, u"modeldata",
  MemberAuthentication(encoding="sha1"), # Only signed with the
     owner's SHA1 digest.
  PublicResolution(),
  DirectDistribution(),
  CommunityDestination(node_count=1), # Reach only one node each
     time.
  MessagePayload(),
  self.check_model, # Checking function.
  self.on_receive_model, # Receiving function.
  delay=0.0) # Delay for sending message.
```

JSON serializer that can handle a wide variety of nested data structures without any extra effort. In the actual message over the wire, the first two bytes specify the length of the message and the rest is the JSON-encoded representation of the model. This solution offers great flexibility. The way we send data on the wire might change in the future.

We have to be very careful when unserializing data that comes over the network as it can contain malicious code. Using our custom unserializer only specific types of data can be introduced, namely those that are subclasses of `GossipMessage`.

As defined by Dispersy [50], a community can govern 4 different policies of each message: Authentication, Resolution, Distribution, Destination. Our `modeldata` message defines these as shown in Listing 3.1. The message is sent to one member in the community, with 0 message sending delay (there is delay between sending messages, though), it uses public resolution and direct distribution. The payload is a `GossipMessage` object which is converted using JSON.

The most important parts of the Gossip Learning core, namely the active and passive threads can be seen in Listing 3.2. These functions have the same semantics as the ones in the Gossip Learning Framework described in Chapter 2.

Listing 3.2: The code of the active and passive threads.

```
def active_thread(self):
  """
  "Active thread", send a message and wait delta time.
  """

  while True:
    self.send_messages([self._model])
    yield DELAY

def on_receive_model(self, messages):
  """
  One or more models have been received from other peers so we
     update and store.
  """
  for message in messages:
    msg = message.payload.message

    assert isinstance(msg, GossipMessage)

    # Database not yet loaded.
    if self._x == None or self._y == None:
      continue

    # Update model.
    msg.update(self._x, self._y)

    # Store model.
    self._model = msg
```

The core of the Gossip Learning Framework is made up of the above mentioned features. These are used to implement a concrete learning protocol like the Adaline perceptron.

3.2.3 Implementing a learning algorithm

To create a specific learning algorithm, one only has to create a subclass of `GossipMessage`, implementing only the `__init__`, `update`, and `predict` functions. These three functions also completely analogous with the Gossip Learning Framework described in Chapter 2.

Listings 3.3 and 3.4 show the actual source code of the Adaline perceptron and Logistic regression, respectively. As previously stated, they are both sublcasses of `GossipLearningModel` and implement the 3 needed functions.

The full source of the implementation can be found at

`http://github.com/csko/Tribler`

3.3 Experimental setup and results

We have tested two algorithms on the *setosa-versicolor* instance of the *Iris* database [20]. These two algorithms are Logistic regression and Adaline perceptron. The Adaline perceptron is described in Chapter 2. We use the regularized Adaline perceptron, so the update rule was changed to the following:

$$w^{(k+1)} = (1 - \eta_k)w^{(k)} + \frac{\eta_k}{\lambda}(y - \langle w^{(k)}, x\rangle)x \tag{3.2}$$

where $\eta_k = \frac{1}{k+1}$.

The best λ regularization parameter we found was 7. Both models have only a w optimization parameter. It is easy to handle the bias term in the framework. In our implementation, Logistic regression uses the bias term, while Adaline perceptron does not. Initially, $w = 0$. The delay between sending messages was 2 seconds.

The database contains 100 examples of which 90 are used as training examples and 10 as testing examples. These examples were spread amongst all the peers so that each peer has one example locally. Tribler uses public-key cryptography with elliptic curves [9] for authentication and authorization. We created a community by generating a master public/private key-pair (`Tribler/Core/dispersy/crypto.py`). After that, we created 90 peers again by creating 90 public/private key-pairs (`Tribler/Core/dispersy/genkeys.py`).

Our experimental scripts started the 90 peers simultaneously using different port and member ID settings (`startExperiment.sh`), initializing each of them with a different local labeled training example (x and y). These were not complete Tribler instances, but only the so-called "scripts" (see `Tribler/community/gossiplearningframework/script.py`). This way we could do our experiments without starting the Tribler GUI.

Each peer's script is logging the timestamp and the current model prediction 0-1 error over the whole testing dataset. These data are aggregated (`result.py`) and then plotted (`plot.sh`).

Figure 3.2 shows the experimental results for the Adaline perceptron and the Logistic regression, that is, the minimum, average, and maximum prediction error over every node over the whole testing set. We can see that both algorithms converge to an error of 0 over the whole network, however, for Adaline perceptron it takes about 2000 seconds, whereas Logistic regression only needs about 300 seconds.

As a reference, Figure 3.3 shows the results for our simulations with Peersim within the No failure scenario (see Chapter 2). These plots display trends that are quite similar to those obtained in our experiments.

3.4 Listings

In this section we present some of the longer code listings.

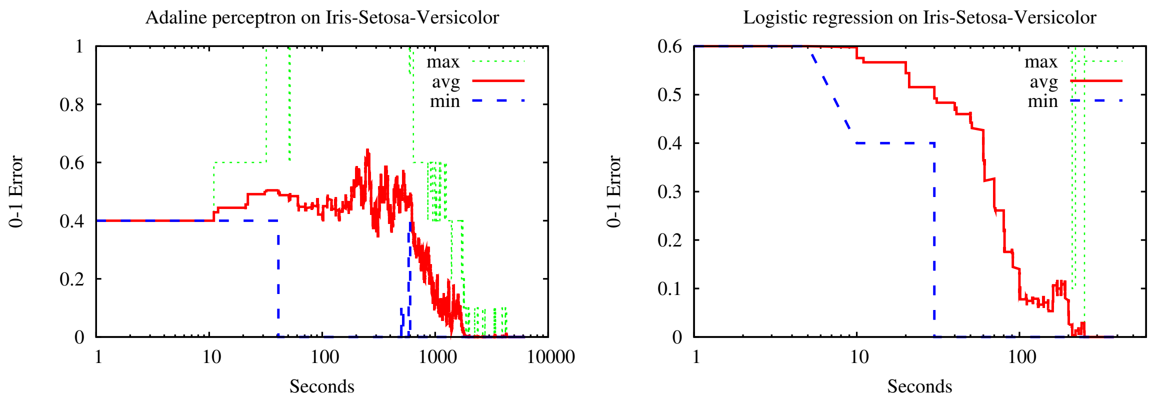

Figure 3.2: Experimental results for Adaline perceptron (left) and Logistic regression (right).

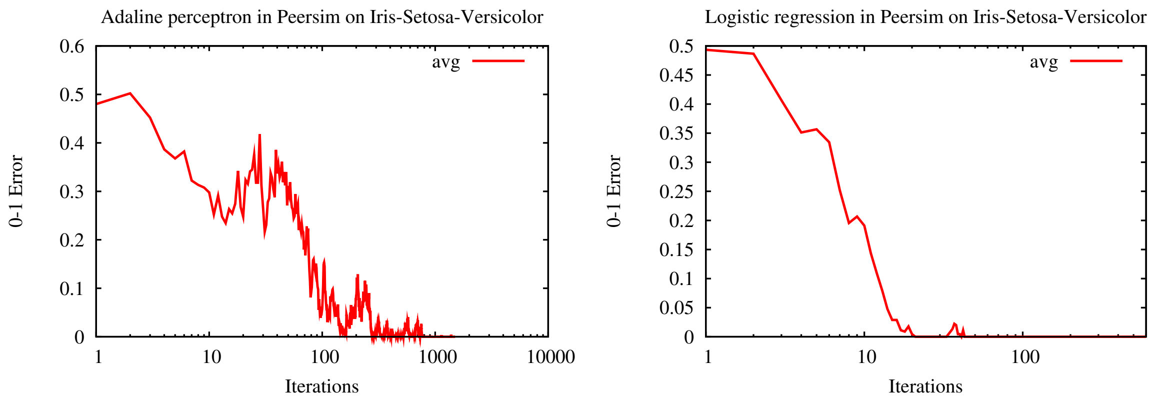

Figure 3.3: Simulation results for Adaline perceptron (left) and Logistic regression (right).

Listing 3.3: The Adaline perceptron model implementation code.

```
class AdalinePerceptronModel(GossipLearningModel):

    def __init__(self):
        super(AdalinePerceptronModel, self).__init__()

        # Initial model
        self.w = [0, 0, 0, 0]
        self.age = 0

    def update(self, x, y):
        """Update the model with a new training example."""
        # Set up some variables.
        x = x[1:] # Remove the bias term.
        label = -1.0 if y == 0 else 1.0 # Remap labels.

        self.age = self.age + 1
        rate = 1.0 / self.age
        lam = 7

        # Perform the Adaline update: w_{i+1} = (1-eta) * w_i + eta/
           lam * (y - w_i' * x) * x.
        wx = sum([wi * xi for (wi,xi) in zip(self.w, x)])
        self.w = [(1-rate) * self.w[i] + rate / lam * (label - wx) *
            x[i] for i in range(len(self.w))]

    def predict(self, x):
      x = x[1:] # Remove the bias term.

      # Calculate w' * x.
      wx = sum([wi * xi for (wi,xi) in zip(self.w, x)])

      # Return sign(w' * x).
      return 1 if wx >= 0 else 0
```

Listing 3.4: The Logistic regression model implementation code.

```
class LogisticRegressionModel(GossipLearningModel):

    def __init__(self):
        super(LogisticRegressionModel, self).__init__()

        # Initial model
        self.w = [0, 0, 0, 0, 0]
        self.age = 0

    def update(self, x, y):
        """Update the model with a new training example."""
        # Set up some variables.
        label = 0.0 if y == 0 else 1.0

        self.age = self.age + 1
        lam = 0.0001
        rate = 1.0 / (self.age * lam)

        # Calculate the probability for this instance.
        prob = self.gx(x)
        err = label - prob

        # Compute the new w value.
        self.w = [(1.0 - rate * lam) * self.w[i] - rate * err * x[i]
            for i in range(len(self.w))]

    def predict(self, x):
        # Find the most likely class.
        pos = self.gx(x)

        if pos > 0.5:
          return 1
        else:
          return 0

    def gx(self, x):
        """Calculate P(Y=1 | X=x, w) = 1 / (1 + e^(w'x))."""
        # Normalization
        x = [x[i]/sum(x) for i in range(len(x))]

        # Calculate w'x.
        wx = sum([wi * xi for (wi,xi) in zip(self.w, x)])

        # exp() can't handle too high or too low parameters
        if wx > 114:
            return 1e-50
        elif wx < -112:
            return 1.0 - 1e-50
        else:
            return 1.0 / (1.0 + exp(wx))
```

Chapter 4

Summary and Further Research Questions

In this deliverable we have described the GOLF framework, that is designed to support a wide range of machine learning algorithms over a dynamic fully distributed system relying only on the peer sampling service. To instantiate the framework, an online algorithm needs to be implemented, along with an optional (but recommended) merging algorithm. The framework does not move, share, or collect data directly. Only machine learning models are moved, which makes it very difficult to infer information about individual data records.

With relatively little effort, the framework is suitable for implementing algorithms that were envisioned at the start of the QLectives project (recommendation and ranking) as well as algorithms that were not envisioned to become part of WP2.3 (spam filtering, vandalism detection). The GOLF framework has been implemented in the P2P QLectives platform and the implementation has been validated on synthetic data.

Our future work involves multiple goals. The first is to study, characterize, and enhance the privacy preserving features of the framework more rigorously. Second, we plan to implement various instantiations including recommender systems. Third, we plan to contribute to the completion of WP4.4, via supporting IRT during the testing phase of the metadata filtering algorithms.

Bibliography

[1] G. Adomavicius and E. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Trans. on Knowledge and Data Engineering*, 17:734–749, 2005.

[2] N. Andrade, T. Vinkó, and J. Pouwelse. Qmedia v2 - short report. Deliverable D.4.3.2, QLectives Project, 2011.

[3] H. Ang, V. Gopalkrishnan, S. Hoi, and W. Ng. Cascade RSVM in peer-to-peer networks. In W. Daelemans, B. Goethals, and K. Morik, editors, *Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, volume 5211 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2008.

[4] H. Ang, V. Gopalkrishnan, W. Ng, and S. Hoi. Communication-efficient classification in P2P networks. In W. Buntine, M. Grobelnik, D. Mladenic, and J. Shawe-Ta ylor, editors, *Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, volume 5781 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2009.

[5] H. Ang, V. Gopalkrishnan, W. Ng, and S. Hoi. On classifying drifting concepts in P2P networks. In J. Balcázar, F. Bonchi, A. Gionis, and M. Sebag, editors, *Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, volume 6321 of *Lecture Notes in Computer Science*, pages 24–39. Springer, 2010.

[6] X. Bai, M. Bertier, R. Guerraoui, A.-M. Kermarrec, and V. Leroy. Gossiping personalized queries. In *Proceedings of the 13th International Conference on Extending Database Technology (EBDT'10)*, 2010.

[7] A. Bakker, E. Ogston, and M. 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.

[8] E. Bauer and R. Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, 36(1):105–139, 1999.

[9] I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic curves in cryptography*. Cambridge University Press, New York, NY, USA, 1999.

[10] L. Bottou. The tradeoffs of large-scale learning, 2007. tutorial at the 21st Annual Conference on Neural Information Processing Systems (NIPS), `http://leon.bottou.org/talks/largescale`.

[11] L. Bottou and Y. LeCun. Large scale online learning. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.

[12] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, 52(6):2508–2530, 2006.

[13] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[14] L. Breiman. Pasting small votes for classification in large databases and on-line. *Machine Learning*, 36(1-2):85–103, July 1999.

[15] S. Buchegger, D. Schiöberg, L.-H. Vu, and A. Datta. PeerSoN: P2P social networking: early experiences and insights. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems (SNS'09)*, pages 46–52, New York, NY, USA, 2009. ACM.

[16] S. G. Cheetancheri, J. M. Agosta, D. H. Dash, K. N. Levitt, J. Rowe, and E. M. Schooler. A distributed host-based worm detection system. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense (LSAD'06)*, pages 107–113, New York, NY, USA, 2006. ACM.

[17] M. Condliff, D. Lewis, D. Madigan, and C. Posse. Bayesian mixed-effects models for recommender systems. In *Proc. ACM SIGIR*, volume 99. Citeseer, 1999.

[18] S. Datta, K. Bhaduri, C. Giannella, R. Wolff, and H. Kargupta. Distributed data mining in peer-to-peer networks. *IEEE Internet Computing*, 10(4):18–26, July 2006.

[19] Diaspora. https://joindiaspora.com/.

[20] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[21] I. Guyon, A. B. Hur, S. Gunn, and G. Dror. Result analysis of the nips 2003 feature selection challenge. In *Advances in Neural Information Processing Systems 17*, pages 545–552. MIT Press, 2004.

[22] C. Hall and A. Carzaniga. Uniform sampling for directed P2P networks. In H. Sips, D. Epema, and H.-X. Lin, editors, *Euro-Par 2009*, volume 5704 of *Lecture Notes in Computer Science*, pages 511–522. Springer, 2009.

[23] P. Han, B. Xie, F. Yang, J. Wang, and R. Shen. A novel distributed collaborative filtering algorithm and its implementation on p2p overlay network. In H. Dai, R. Srikant, and C. Zhang, editors, *Advances in Knowledge Discovery and Data Mining*, volume 3056 of *LNCS*, pages 106–115. Springer, 2004.

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

[25] S. Isaacman, S. Ioannidis, A. Chaintreau, and M. Martonosi. Distributed rating prediction in user generated content streams. In *Proceedings of the fifth ACM conference on Recommender systems*, RecSys '11, pages 69–76, New York, NY, USA, 2011. ACM.

[26] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems*, 23(3):219–252, August 2005.

[27] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3):8, August 2007.

[28] V. King and J. Saia. Choosing a random peer. In *Proceedings of the 23rd annual ACM symposium on principles of distributed computing (PODC'04)*, pages 125–130. ACM Press, 2004.

[29] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[30] W. Kowalczyk and N. Vlassis. Newscast EM. In L. K. Saul, Y. Weiss, and L. Bottou, editors, *17th Advances in Neural Information Processing Systems (NIPS)*, pages 713–720, Cambridge, MA, 2005. MIT Press.

[31] P. Luo, H. Xiong, K. Lü, and Z. 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.

[32] J. Ma, L. K. Saul, S. Savage, and G. 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.

[33] M. Medo. Simulations of quality emergence. Deliverable D.1.3.1, QLectives Project, 2010.

[34] R. Ormándi, I. Hegedűs, K. Csernai, and M. Jelasity. Towards inferring ratings from user behavior in BitTorrent communities. In *Proceedings of the 6th International Workshop on Collaborative Peer-to-Peer Systems (COPS) at WETICE'10*, pages 217–222. IEEE Computer Society, 2010.

[35] R. Ormándi, I. Hegedűs, R. Farkas, and M. Jelasity. Novel balanced feature representation for Wikipedia vandalism detection task. In *CLEF 2010 Labs and Workshops*, 2010. `http://clef2010.org/index.php?page=pages/proceedings.php`.

[36] R. Ormándi, I. Hegedűs, and M. Jelasity. Overlay management for fully distributed user-based collaborative filtering. In P. D'Ambra, M. Guarracino, and D. Talia, editors, *Euro-Par 2010*, volume 6271 of *Lecture Notes in Computer Science*, pages 446–457. Springer-Verlag, 2010.

[37] Y.-J. Park and A. Tuzhilin. The long tail of recommender systems and how to leverage it. In *Proceedings of the 2008 ACM conference on Recommender systems*, RecSys '08, pages 11–18, New York, NY, USA, 2008. ACM.

[38] PeerSim. http://peersim.sourceforge.net/.

[39] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, and H. J. Sips. TRIBLER: a social-based peer-to-peer system. *Concurrency and Computation: Practice and Experience*, 20(2):127–138, 2008.

[40] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *Proc. 1994 ACM Conf. on Computer supported cooperative work (CSCW '94)*, pages 175–186. ACM, 1994.

[41] L. Rokach. Ensemble-based classifiers. *Artificial Intelligence Review*, 33(1):1–39, 2010.

[42] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter. Pegasos: primal estimated sub-gradient solver for SVM. *Mathematical Programming B*, 2010.

[43] M. Shang, C. Jin, T. Zhou, and Y. Zhang. Collaborative filtering based on multi-channel diffusion. *Physica A: Statistical Mechanics and its Applications*, 388(23):4867–4871, 2009.

[44] S. Siersdorfer and S. Sizov. Automatic document organization in a p2p environment. In Lalmas, M et al., editor, *Advances in Information Retrieval*, volume 3936 of *LNCS*, pages 265–276. Springer, 2006.

[45] S. Siersdorfer and S. Sizov. Automatic document organization in a P2P environment. In M. Lalmas, A. MacFarlane, S. Rüger, A. Tombros, T. Tsikrika, and A. Yavlinsky, editors, *Advances in Information Retrieval*, volume 3936 of *Lecture Notes in Computer Science*, pages 265–276. Springer, 2006.

[46] D. Stutzbach, R. Rejaie, N. Duffield, S. Sen, and W. Willinger. On unbiased sampling for unstructured peer-to-peer networks. *IEEE/ACM Transactions on Networking*, 17(2):377–390, April 2009.

[47] G. Takács, I. Pilászy, B. Németh, and D. Tikk. Scalable collaborative filtering approaches for large recommender systems. *J. Mach. Learn. Res.*, 10:623–656, June 2009.

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

[49] R. van Renesse, K. P. Birman, and W. 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.

[50] T. Vinkó, N. Andrade, B. Schoon, and J. Pouwelse. Qlectives platform v2. Deliverable D.4.1.2, QLectives Project, 2011.

[51] B. Widrow and M. E. Hoff. Adaptive switching circuits. In *1960 IRE WESCON Convention Record, Part 4*, pages 96–104, New York, 1960. IRE.

[52] T. Zhou, Z. Kuscsik, J. Liu, M. Medo, J. Wakeling, and Y. Zhang. Solving the apparent diversity-accuracy dilemma of recommender systems. *Proceedings of the National Academy of Sciences*, 107(10):4511, 2010.