University of Szeged

Department of Informatics

Distributed Machine Learning Using the Tribler Platform

Master's Thesis

Author:
Kornél Csernai
Software Information Technologist (MSc.)

Advisor:
Dr. Márk Jelasity
Senior Research Scientist

Szeged
2012

Contents

Problem Specification

Nowadays data mining involves working with large amounts of data which often cannot be processed on a few central servers, rather, they must be handled in a fully distributed manner. The author of this thesis overviews the machine learning and distributed computing background and gossiping techniques, especially the Gossip Learning Framework.

The main contribution of this thesis is the implementation and evaluation of a fully distributed algorithm in a real world application.

Summary

We overviewed machine learning techniques and peer-to-peer networks as well as Tribler, a popular BitTorrent-based fully distributed social-based content sharing platform and its technical details, e.g. the Dispersy permission system. Working together with the developers of Tribler, we realized that there is a great need for machine learning solutions, such as spam filtering or vandalism detection.

We decided to implement the Gossip Learning Framework in Tribler, which is a robust, asynchronous, gossip based protocol that can withstand high churn and failure rates, making it ideal for peer-to-peer networks. In this setting, each peer trains on their local training examples (which could be very few) and pass along the trained models to their neighbors. This keeps the network complexity low as well as provide some privacy while being able to collectively learn the structure of the data.

We have investigated ways of integrating this protocol into Tribler, and we concluded that the best way to do this is to create a so-called community within Tribler. To validate our implementation, we loaded up two databases into the Tribler network and were able to reproduce previous simulation results.

Tools used: GoLF (Peersim) peer-to-peer simulator written in Java, run on a high performance Linux server. The Tribler community was written in Python2. All of the source code and tools are publicly available.

Keywords: *machine learning, distributed systems, gossip protocols, Tribler*

Tartalmi összefoglaló

A gépi tanulási módszerek és peer-to-peer hálózatok áttekintését követően a Tribler nevű népszerű BitTorrent alapú teljesen elosztott szociális tartalommegosztó rendszerrel, és annak technikai részleteivel foglalkoztunk (így például a Dispersy jogosultság rendszerrel). A Tribler fejlesztőivel együttműködve beláttuk, hogy a rendszernek nagy szüksége lenne gépi tanulási megoldásokra, mint például a spam-szűrés vagy a vandalizmus detekció.

Úgy döntöttünk, hogy a Gossip Learning Framework nevű robosztus, aszinkron, pletyka alapú rendszert implementáljuk a Tribleren belül. Ez egy kiválóan alkalmas peer-to-peer rendszerek, mely képes ellenállni a felhasználók gyakori ki- és belépésének illetve gyakori meghibásodásoknak. A feladathoz tartozik, hogy a csomópontok csupán néhány lokális tanulópéldával rendelkeznek és az ezeken tanított modelleket küldik a szomszédaiknak. Ez alacsonyan tartja a hálózati forgalmat és bizonyos magánéleti védelmet is biztosít, miközben lehetőség nyílik kollektíven tanulni az adatok struktúráját.

Megvizsgáltuk a protokoll Triblerbe integrálásának lehetőségeit, és arra jutottunk, hogy a legjobb megoldás, ha egy úgynevezett közösséget (*community*) hozunk létre a Tribler rendszerén belül. Annak érdekében, hogy meggyőződjünk az implementáció helyességéről, két tanító adatbázissal feltöltöttük a Tribler hálózatát és sikerült reprodukálni a korábbi szimulációs eredményeket.

Felhasznált eszközök: GoLF (Peersim) Java nyelven írt peer-to-peer szimulátor, amelyet egy nagy teljesítményű Linux szerveren futtattunk. A Tribler közösség kódja Python2 nyelven íródott. Az összes forráskód és a felhasznált eszközök nyilvánosak.

Kulcsszavak: *gépi tanulás, elosztott rendszerek, pletyka protokollok, Tribler*

Introduction

Nowadays, the data available on the internet is increasing at a high pace, especially because people and machines generate data at the same time. Machine learning over fully distributed data in peer-to-peer (P2P) applications is an interesting problem. For example, the case of social network profiles, mobile networks sensor readings could benefit from machine learning on data that is fully distributed.

In the extreme case, we have very few (maybe only one) training example available on each peer, which means that we can not learn a model locally. Instead, we learn models in an online way on each peer and send the models to other peers. The size of the models can be considerably smaller than a few training examples, also, it provides some sense of privacy.

At this time, there are not many known deployed systems that use this kind of machine learning. In this thesis we discuss a machine learning framework and show how one can implement it in a real-life application, such as the Tribler peer-to-peer content sharing platform.

We will focus on linear models, e.g. Logistic regression. We then analyze the results of the implementation through various tests and conclude that they concur with previous simulation results.

The thesis is structured as follows: In Chapter 1, we will talk about some of the basic peer-to-peer design principles. The chapter will also present an introduction to machine learning learning while focusing on supervised learning, and more specifically, stochastic gradient descent. After that, Chapter 2 will discuss the Gossip Learning Framework (GoLF), which is the subject of the implementation in this thesis. Next, in Chapter 3, we introduce the target platform called Tribler along with a few technical details. Following that, in Chapter 4, we dive into the actual implementation details. In Chapter 5 we show that our implementation works and compare it to previous simulation results.

Chapter 1

Preliminaries

This chapter gives an introduction to the two fields related to this work: Peer-to-Peer Networks and Machine Learning. Some notations will be introduced as well. This will hopefully help readers unfamiliar with these two areas understand the remainder of this thesis. For a complete overview of these fields, please refer to proper textbooks and resources which will be referenced later on.

1.1 Peer-to-Peer Systems

Nowadays with the increasing usage of personal computers and mobile devices connected to the internet, it is important to build systems that enable their users to efficiently access data. Some of the most popular applications are real-time and on-demand video and audio streaming (YouTube, Netflix, justin.tv, ...), Voice over IP (Skype), social networks (Facebook, Twitter, Google+, ...), file sharing protocols (private and public BitTorrent communities), web searching, cloud services (Amazon Web Services). In the last few years the number of smartphones and tablets has increased tremendously.

1.1.1 The peer-to-peer paradigm

One way to structure the workload in a distributed computing environment is the *client-server* architecture, which has been used effectively for a long time. In this setting, there are one or more dedicated servers to which clients connect through the network. The servers provide some kind of resoruce that the clients are interested in. For example, this could be a web, video, or email service. Every client depends on the servers, and the

servers take all the workload. This is a big disadvantage of the client-server paradigm.

Instead, an other way would be to have every computer act as a server and a client at the same time, which gives us the *peer-to-peer* architecture. We will call the participating computers as *node*s, or *peer*s. Each node can serve others and request resources simultaneously. This eliminates the single point of failure. However, as a consequence, algorithms become more complicated.

1.1.2 Basic concepts

In a P2P network, each node is connected to a set of other peers, we call them *neighbors*. The (virtual) networks of these connections are called *overlay network*s.

Network connections on the internet could use the TCP or UDP protocols. TCP is stateful, has error detection, is mainly used for sending documents and other critical data. On the other hand, UDP is stateless, which makes it suitable for video and audio transmission.

Connectability

Nodes are often connected to the internet through Network Address Translation and/or firewalls which might be configured not to accept incoming connections. We call them *unconnectable*. This makes it difficult to reach out to these peers, they can, however, reach other, *connectable* peers. The connectability problem has made designing some P2P applications more difficult, as the ratio of unconnectable peers on the internet is quite high, can range between 35% and 90% [6, 12] depending on the application and geographical location. There are methods to exploit some NAT devices, e.g. the method often referred to as *NAT puncturing* or *NAT traversal*. Generally, protocols using UDP can more effectively go around NATs than protocols built on TCP [8].

Churn

In practice, existing peers disconnect and new peers connect all the time. This is called *churn*. Session lengths can typically be approximated with log-normal distribution [19].

Even if the protocol requires the peer to inform others when they disconnect, this might not always happen because of network and system failures, or misbehavior. A well-designed protocol takes this into consideration. Protocols must endure the massive arrival and departure of peers as well.

	Upstream		Downstream		Aggregate	
Rank	Application	Share	Application	Share	Application	Share
1	BitTorrent	52.01%	Netflix	29.70%	Netflix	24.71%
2	HTTP	8.31%	HTTP	18.36%	BitTorrent	17.23%
3	Skype	3.81%	YouTube	11.04%	HTTP	17.18%
4	Netflix	3.59%	BitTorrent	10.37%	YouTube	9.85%
5	PPStream	2.92%	Flash Video	4.88%	Flash Video	3.62%
6	MGCP	2.89%	iTunes	3.25%	iTunes	3.01%
7	RTP	2.85%	RTMP	2.92%	RTMP	2.46%
8	SSL	2.75%	Facebook	1.91%	Facebook	1.86%
9	Gnutella	2.12%	SSL	1.43%	SSL	1.68%
10	Facebook	2.00%	Hulu	1.09%	Skype	1.29%
	Top 10	83.25%	**Top 10**	84.95%	**Top 10**	82.89%

SOURCE: SANDVINE NETWORK DEMOGRAPHICS

sandvine

Figure 1.1: Top internet applications in North America in 2011 [16].

1.1.3 Significance

Most of the peer-to-peer (P2P) traffic is file-sharing, video-streaming, and content delivery, in general. According to a study done by Sandvine[16], in the spring of 2011, P2P file-sharing (BitTorrent) was responsible for 52.01% of all upstream traffic in North America in peak periods (see Figure 1.1). Skype has a good chunk of upstream traffic as well, which uses P2P too. The report concludes that P2P file-sharing is dropping, and Real-Time Entertainment and Mobile traffic is on the rise. While the latter two are not necessarily P2P applications, they could still be done in a distributed way.

1.1.4 BitTorrent

BitTorrent is one of the most popular P2P file-sharing applications. It was first introduced in 2001 by Bram Cohen, a publication followed in 2003 [4], and it has been in the focus of research and media the following years. Its main advantage is that it scales well with the number of users and it is user friendly.

A *torrent* is a metadata file format that describes sets of files to be shared. It contains information such as file sizes, CRC, and tracker addresses. The files are split into fixed-size *chunks* or *pieces*, which are transferred in blocks. A group of peers sharing the

same torrent is called a *swarm*. Each peer is downloading and uploading simultaneously. In the original protocol, there is a central server called the *tracker*. Each peer contacts the tracker in order to get a set of random peers from the same swarm. This creates a single point of failure, which is mitigated by Distributed Hash Tables for instance. The original client was written in Python, but since then numerous client implementations have surfaced which have a wide range of interfaces (GUI, console, web, mobile). One of these BitTorrent clients is called Tribler, which is in the focus of the implementation in this thesis.

1.2 Machine Learning

Machine Learning grew out of artificial intelligence. It is a field of study that gives computers the ability to learn without being explicitly programmed (Arthur Samuel, 1959.). Tom Mitchell defined Machine Learning as the follows [11]:

> 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.

Machine Learning has been effectively used to solve many interesting problems: spam detection, search engines, data mining, recommendation systems, natural language processing, speech recognition, computer vision, robotics, games, and much more.

In the remainder of this chapter we will introduce a few learning algorithms which we will use later and some basic methods for applying Machine Learning algorithms. The resources used here are [2, 13, 7].

1.2.1 Problem types

Machine Learning problems can *roughly* be categorized into a few major types:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

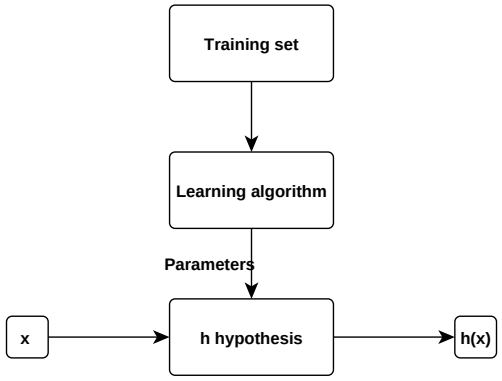

Figure 1.2: The supervised learning process.

When we first encounter a problem, we have to determine which category it belongs to. The three categories employ different techniques and algorithms. Some of these will be presented briefly. This work focuses on supervised learning, but the other two ways have also been successfully used in P2P networks.

Supervised learning

Let $x \in \mathcal{X}$ be an input *feature* vector and $y \in \mathcal{Y}$ a *target* vector. The *training set* comprises of m training examples, $S = \left\{(x, y)^{(i)}\right\}_{i=1}^{m} \subseteq (\mathcal{X} \times \mathcal{Y})$. The goal is to find a $h : \mathcal{X} \mapsto \mathcal{Y}$ *hypothesis* function that *predicts* the correct corresponding y value based on an x value. After successfully finding a hypothesis, we will use h to predict the $y \in \mathcal{Y}$ values for some $x \in \mathcal{X}$ we have not seen before (which should come from the same distribution as the previously seen training examples).

In the case of online learning, we have to make a prediction for some x values before being able to see every training example. Online learning will be very useful in the implementation of supervised learning algorithms in distributed networks.

When the y target value is a continuous variable, we are dealing with a *regression* problem, whereas when y is discrete, we have a *classification* problem. In this case, y is called the *label*. As a special case, when we only have two values for y, for example $\mathcal{Y} = \{0, 1\}$, we have a binary classification problem. The perceptron algorithm, for instance, uses $\mathcal{Y} = \{-1, 1\}$.

Well known methods and algorithms:

- Linear Regression

- Logistic Regression
- Perceptron
- Support Vector Machines
- Bayesian Decision methods
- Decision Trees
- Artificial Neural Networks
- Hidden Markov Models

Unsupervised learning

This case is similar to Supervised learning, except there are no y target values present in the training set. The goal is to find clusters, or other structures in the data.

Well known methods and algorithms:

- K-means clustering
- Expectation Maximization
- Principal Component Analysis

Reinforcement learning

In this setting, it is hard to determine if an action is right or not, we do not have a training set. Rather, the algorithms are provided with a reward function which punishes or rewards the agent based on its actions.

Reinforcement learning is based on Markov Decision Processes (*MDP*s). Some of the methods to work with MDPs are *value iteration* and *policy iteration*.

1.2.2 Linear regression

First, let us take a look at a simple supervised machine learning algorithm for real-valued prediction, the linear regression. After that, we will discuss three algorithms that we use in our implementation.

Let us consider a real valued n-dimensional regression problem, that is, we are looking for a parameterized hypothesis $h_\theta : \mathbb{R}^n \mapsto \mathbb{R}$, where $\theta = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{pmatrix} \in \mathbb{R}^{n+1}$.

In the case of linear regression, the hypothesis h is a linear function, a hyperplane:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n = \sum_{i=0}^{n} \theta_i x_i = \theta^T x$$

where $x_0 = 1$ is the *bias term*, by convention. This can be extended to polynomial regression.

1.2.3 Logistic regression

Let us now consider a real valued n-dimensional binary classification problem, that is, we are looking for a parameterized hypothesis $h_\theta : \mathbb{R}^n \mapsto \{0, 1\}$, where θ is the same as above. We could use a separating hyperplane for this case as well, but that would perform really poorly on a binary class. We will use the sigmoid form instead:

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

where

$$g(x) = \frac{e^x}{e^x + e^0} = \frac{1}{1 + e^{-x}}$$

is the *sigmoid function* shown in Figure 1.3.

1.2.4 Gradient descent

In this subsection we will introduce an iterative method that solves both linear regression and logistic regression. For linear regression, we use the *least squares cost function*:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2.$$

For logistic regression, we use a slightly different cost function:

$$J(\theta) = \sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))$$

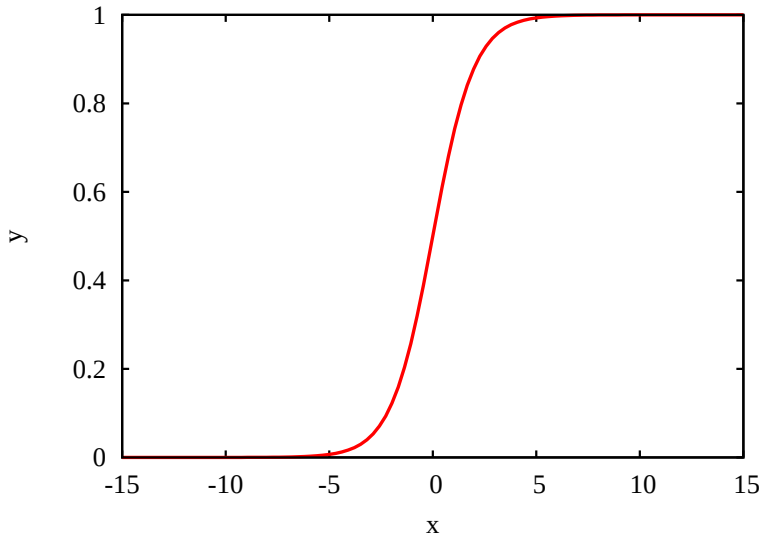

Figure 1.3: The sigmoid function $g(x) = \frac{1}{1+e^{-x}}$.

The idea is that we want to minimize errors on the training set and that will lead us to the best hypothesis. That is, we have to minimize the cost function. One such optimization algorithm is the *gradient descent* algorithm. It starts off with an arbitrarily defined θ parameter, and works its way to the optimal θ^*, each step possibly decreasing the cost function.

In each step, it modifies $\theta_j, j = 0, \ldots, n$ simultaneously using the following update rule:

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

where $\alpha \in \mathbb{R}^+$ is the learning rate.

The partial derivative tells us the direction the cost function is increasing the most. Since we are minimizing the cost function, we want to move θ to the direction opposite to the derivative. If the derivative is positive, the cost function is increasing, and so we will decrease θ, and as a result, decreasing $J(\theta)$. In case the derivative is negative, the cost function is decreasing, and so we will be increasing θ, and as a result, decrease $J(\theta)$.

For both linear regression and logistic regression, we get the following partial derivative of $J(\theta)$:

Algorithm 1 Batch gradient descent algorithm.

1: Choose initial θ_j and α.
2: **repeat**
3: $\quad \theta_j \leftarrow \theta_j - \alpha \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$ (for $j = 0, \ldots, n$)
4: **until** convergence

Algorithm 2 Stochastic gradient descent algorithm.

Shuffle the dataset.
for $i = 1, \ldots, m$ **do**

$$\theta_j \leftarrow \theta_j - \alpha (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

(for $j = 0, \ldots, n$)

end for

$$\frac{\partial}{\partial \theta_j} J(\theta) = \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Using this, we can construct our first gradient descent algorithm, as shown in Algorithm 1. Even though the gradient descent only finds one local optimum, it will be a suitable method for optimizing $J(\theta)$, because it is a convex quadratic function which has only one local optimum, which is a global optimum as well. Figure 1.4 shows an example of a multi-dimensional quadratic function.

This version is called *batch gradient descent* and it considers every training example for each iteration. While this works in some cases, we would rather not look at each training example. If the training set is too large, or when it is not feasible to access every training example, this method will not work well.

Another to work around this is to use a subset of the training set, $Q \subseteq S$ in each iteration. This is called *mini-batch gradient descent* and has the following update rule:

$$\theta_j = \theta_j - \alpha \sum_{(x,y) \in Q} (h_\theta(x) - y) x_j$$

(for $j = 0, \ldots, n$)

As a special case, we can update θ for each training example. This is called *stochastic gradient descent* (SGD) and has the following update rule shown in Algorithm 2:

The stochastic gradient descent will play a key role in the following chapters when working with Peer-to-Peer systems. We will assume that every peer has only one local

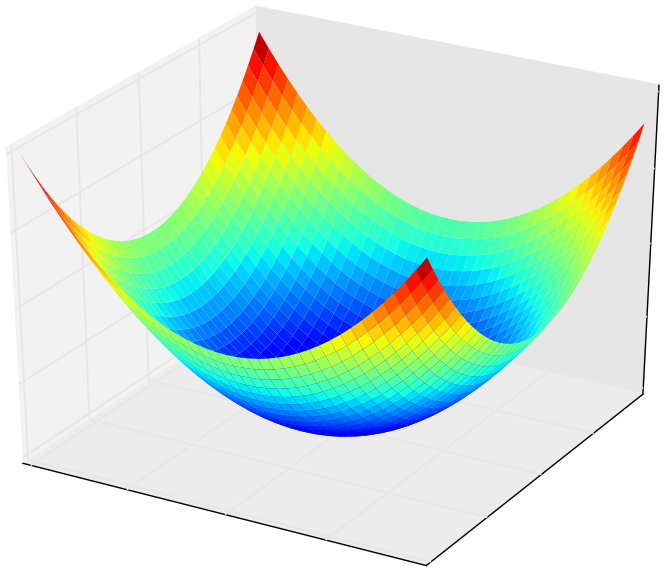

Figure 1.4: Multi-dimensional quadratic function.

training example.

Choosing α

The gradient descent method converges, given we chose an appropriate value for α. Choosing a good value for the learning rate depends on the data we have at hand. If too small, the algorithm might converge too slowly. If too large, however, the algorithm might even diverge. One can fine tune the value of α with dataset samples.

The series of α should be divergent, and the power sum of α should be convergent, i.e. $\sum \alpha = \infty, \sum \alpha^k < \infty \ (k > 1)$. For stochastic gradient descent, we usually choose an α that is a decreasing function of the iteration counter t, it can be as simple as $\alpha_t = \frac{1}{t}$.

1.2.5 Adaline Perceptron and Pegasos

In this section we give a brief introduction to two other algorithms that can be plugged into the Stochastic gradient optimization framework.

Adaline Perceptron

The Adaline perceptron [21] is a one-layer neural network developed in 1960. Consider the cost function

$$J(\theta) = \frac{1}{2}(y - h_\theta(x))^2$$

for the binary classification problem, where $\mathcal{X} = \mathbb{R}^n, \mathcal{Y} = \{-1, 1\}$.

The gradient at θ for x is

$$\frac{\partial}{\partial\theta} J(\theta) = -(y - \theta^T x)x$$

which yields the (vectorized) update rule

$$\theta = \theta + \eta(y - \theta^T x)x \tag{1.1}$$

where $\eta = \frac{1}{\alpha t}$ is the learning rate. It is then straightforward to plug this in the SGD framework.

After regularization, the Adaline perceptron update rule becomes the following:

$$\theta = (1 - \eta)\theta + \frac{\eta}{\lambda}(y - \theta^T x)x \tag{1.2}$$

Pegasos

Support Vector Machines (SVM)[5] is a popular method for solving various machine learning tasks. Its optimization problem can be written in two equivalent forms, the primal problem and the dual problem.

Primal problem:

$$\begin{aligned} \min_{w,\xi_i,b} \quad & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\xi_i \\ \text{s.t.} \quad & y_i(w^T x_i - b) \geq 1 - \xi_i, \quad i = 1, \ldots, m \\ & \xi_i \geq 0, \quad i = 1, \ldots, m \end{aligned} \tag{1.3}$$

Algorithm 3 Pegasos algorithm update rule (simplified).

```
η ← 1/(α · t)
if y · w^T x < 1 then
    w ← (1 − ηα)w + ηyx
else
    w ← (1 − ηα)w
end if
```

Dual problem:

$$\begin{aligned} \max_{\alpha} \quad & W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^T x_j \\ \text{s.t.} \quad & 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, m \\ & \sum_{i=1}^{m} \alpha_i y_i = 0 \end{aligned} \tag{1.4}$$

In our setting, we only consider working with the linear kernel version, which is shown in Equations 1.3 and 1.4.

We will use the Primal Estimated sub-GrAdient SOlver for SVM (Pegasos) algorithm[18], which solves the SVM problem in a SGD based approach. It solves the primal problem, however, most SVM algorithms (e.g. SMO) solve the dual problem. The simplified version of this algorithm is shown in Algorithm 3. As usual, x is the feature vector, y is the class label, w is the hyperplane, t is the iteration counter, and η is the learning rate.

1.2.6 Working with supervised algorithms

This section focuses on supervised learning, however, some concepts can be used with other problem types as well. In the remainder of this chapter, we will be considering supervised learning problems.

Advanced evaluation

Previously we introduced the training set on which we build our model. In order to have an idea of the performance of the model, we must evaluate it. This is normally done on a separate *testing set* S_{test}, which is a set of (x, y) pairs. In this simple setting, we first create our hypothesis h on the training set, then for each x in the testing set, we use h to predict

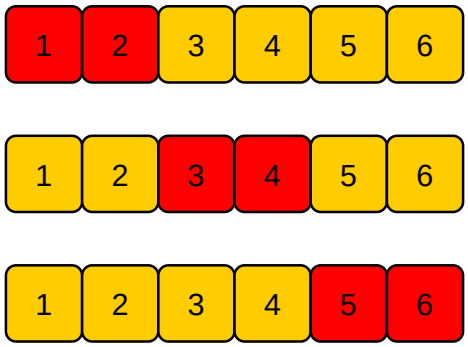

Figure 1.5: Cross-validation with $m = 6, k = 2$.

$h(x)$. After that, we compare $h(x)$ to y to determine if there was an error. The error definition can be application-specific (zero-one error, mean absolute error, etc.). Usually the initial database is split into training and testing sets. The ratio could be 70:30.

k-fold cross-validation is an advanced evaluation method (as described in [13]):

1. Randomly split S into k disjoint subsets of $\lceil m/k \rceil$ training examples each. Call these subsets $S_1, \ldots, S_k$.

2. For $j = 1, \ldots, k$

   Train on $S_1 \cup \cdots \cup S_{j-1} \cup S_{j+1} \cup \cdots \cup S_k$ (that is, train on all the data except S_j) to get hypothesis h_j. Calculate the error of h_j only on S_j.

3. The estimated generalization error of the model is the average error over j.

Figure 1.5 shows an example setup for cross-validation. The special case $k = m$, is called *leave-one-out cross validation*.

Normalization

For many machine learning algorithms, it is very important that we normalize our input feature vectors. Otherwise, the algorithms might converge very slowly, if at all.

Normalization can be done by subtracting the mean and dividing by the variance. One other way to do it is to divide by the range (the difference of the maximum and the minimum). The normalization when predicting should use the parameters (e.g. variance)

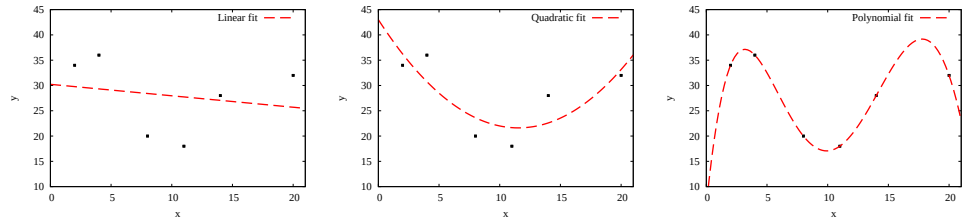

Figure 1.6: A linear (left), a quadratic (middle), and a 5th order polynomial (right) fit of a quadratic function with added noise.

from the normalization on the training set, i.e. we should normalize data using the same operations.

Generalization

For a regression problem, consider some data points and three models, shown in Figure 1.6:

- A linear model (a).
- A 2$^{\text{nd}}$ order polynomial model (b).
- A 5$^{\text{th}}$ order polynomial model (c).

The hypotheses a and c show the two cases of generalization error. The case of a is called *underfitting*. It does reasonably well on the training set, but does not have the generalization power to capture the structure and will perform poorly on new examples. This is called *high bias*.

On the other hand, c is *overfitting* the training set. While it makes very good predictions for the training set (predicting the exact value of each of the 6 points), it too fails to capture the right structure and will perform poorly on new examples. This is called *high variance*.

Often there is a trade-off between bias and variance. We have to balance the number of features. Choosing the right number of features can be done with the help of cross-validation. Examining the learning curve can be useful as well (that is, the function of some parameter of the model and the learning error).

Chapter 2

Gossip Learning Framework

In this section, we combine peer-to-peer systems and machine learning techniques. Specifically, we overview the P2P computational framework called Gossip Learning Framework from [14]. This paper focuses on machine learning for linear models in a P2P network where data is fully distributed. This solution is unique in the sense that the data does not leave the nodes, only models are transmitted through the network. The models essentially perform a random walk. Each node should be able to predict using only locally available data and there should be low network complexity. The proposed generic algorithm is very robust, even in harsh environments of churn and message drop and latency, it performs relatively well.

2.1 Machine learning on fully distributed data

The authors consider the extreme case when every node in the network stores one, and only one data record, so there are as many nodes as data records. For example, this could be a sensor reading.

The data never leaves the nodes, only the models are transmitted. This provides some sense of privacy, as well as robustness. In some cases the local data would be too much to be transferred over the wire. These are really good features and prove to be useful in the applications of mobile phones and social networking.

It is noted that there are algorithms for computing functions over a P2P network when data is distributed: aggregation (sum, minimum, etc.), Expectation Maximization, collaborative filtering. However, it is often assumed that each peer has more than one training data locally and can learn using only that. In this case, model passing would be of less

Algorithm 4 Gossip learning skeleton

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

use.

2.2 Gossip learning skeleton

The proposed method in this paper uses a gossip message passing scheme, as shown in Algorithm 4. The algorithm has an active and a passive component, both of which run on every client in the network. In the default setting, each peer has one *locally available* labeled training example, and a model, e.g. a vector representing hyperplane. The initial model could be the zero vector, which is later improved by the update method. In a more general setting, each node can store a queue of models of some size (modelCache). This allows for advanced schemes such as voting.

The active thread runs in cycles, with Δ time between them (e.g. 10 seconds). The cycles need not start at the same time at each peer. In each cycle a neighboring peer is first selected. It is convenient to use the NEWSCAST gossip algorithm [10] for peer selection purposes, because it can use the gossip messages we are sending anyway, essentially piggybacking on them. It also provides a local set of candidate peers which can be used without any additional network overhead.

After the target peer p has been selected, we send our model to that peer. This method is asynchronous, that is, we do not halt execution until the message has been received. In fact, we do not require that every message arrives and we cannot guarantee that there would be no delay (messages can be out of order as well).

The passive thread, ONRECEIVEMODEL, is called whenever there is an incoming message containing a model. Generally, this method stores the incoming model m in a suitable way. Specifically, we create a new model from m and our last received model

Algorithm 5 CREATEMODEL: three implementations

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

using the createModel method.

The createModel method can be implemented using different strategies, the paper suggests three variants, shown in Algorithm 5. The first one, CREATEMODELRW, essentially guides models through a *random walk*. It only uses the first model, m_1, and updates that using the local training example. The second one, CREATEMODELMU, first merges the two models and then updates the merged model using the training example. Finally, the third one, CREATEMODELUM, updates both models using the local training example, and then merges them.

Algorithm 6 shows a way to initialize and merge models, as well as the update instantiations for the Pegasos and Adaline online learners. The INITMODEL method creates the initial linear model as a zero vector with age 0 and initializes the model queue to have that initial model. One merging strategy in the case of linear models is averaging and taking the maximum of the ages as the new age. For linear models, the sign of the inner product of the separating hyperplane w and query x gives the predicted class.

UPDATEPEGASOS and UPDATEADALINE are the straightforward update method implementations of Algorithm 3 and Equation 1.1, respectively.

Algorithm 6 Pegasos and Adaline model update, prediction, initialization, and merging

```
1:  procedure UPDATEPEGASOS(m)
2:      m.t ← m.t + 1
3:      η ← 1/(α · m.t)
4:      if y · m.w^T x < 1 then
5:          m.w ← (1 − ηα)m.w + ηyx
6:      else
7:          m.w ← (1 − ηα)m.w
8:      end if
9:      return m
10: end procedure
11:
12: procedure UPDATEADALINE(m)
13:     m.w ← m.w + η(y − m.w^T x)x
14:     return m
15: end procedure
16: procedure INITMODEL
17:     lastModel.t ← 0
18:     lastModel.w ← (0, ..., 0)^T
19:     modelCache.add(lastModel)
20: end procedure
21:
22: procedure MERGE(m_1,m_2)
23:     m.t ← max(m_1.t, m_2.t)
24:     m.w ← (m_1.w + m_2.w)/2
25:     return m
26: end procedure
27:
28: procedure PREDICT(x)
29:     w ← modelCache.freshest()
30:     return sign(w^T x)
31: end procedure
```

Chapter 3

The Tribler Architecture

Now that we know about machine learning in distributed systems, let us discuss a real-world example of a peer-to-peer network that we will build our protocols on.

Tribler is a social Peer-to-Peer content sharing platform. It can be thought of as a BitTorrent [4] client, but with many additional features. One main advantage of using Tribler is that it is fully decentralized, meaning that there is no single point of failure (whereas for traditional BitTorrent, the tracker is a single point of failure). Bootstrapping is done by the use of superpeers. Everyone can become a bootstrap peer, see `http://www.tribler.org/trac/wiki/BootstrapTribler`. Tribler has gained a lot of attention lately due to its distributed nature. For detailed statistics, see `http://statistics.tribler.org/`.

Another interesting feature is the integrated media player. This allows playing video files inside Tribler, even when the video file is split into multiple RAR files. Furthermore, Tribler can prioritize the first blocks of the torrent in order to be able to stream the video even before it is completely downloaded.

Tribler offers great searching functionalities. The search is also distributed, there is no need for a tracker. Content is structured into channels (or, communities), which we will cover in detail in Subsection 3.2.2.

Tribler is the product of many years of scientific work done at the Delft University of Technology, and is funded by the European Union 7^{th} Framework Research Programme (P2P-Next, QLectives).

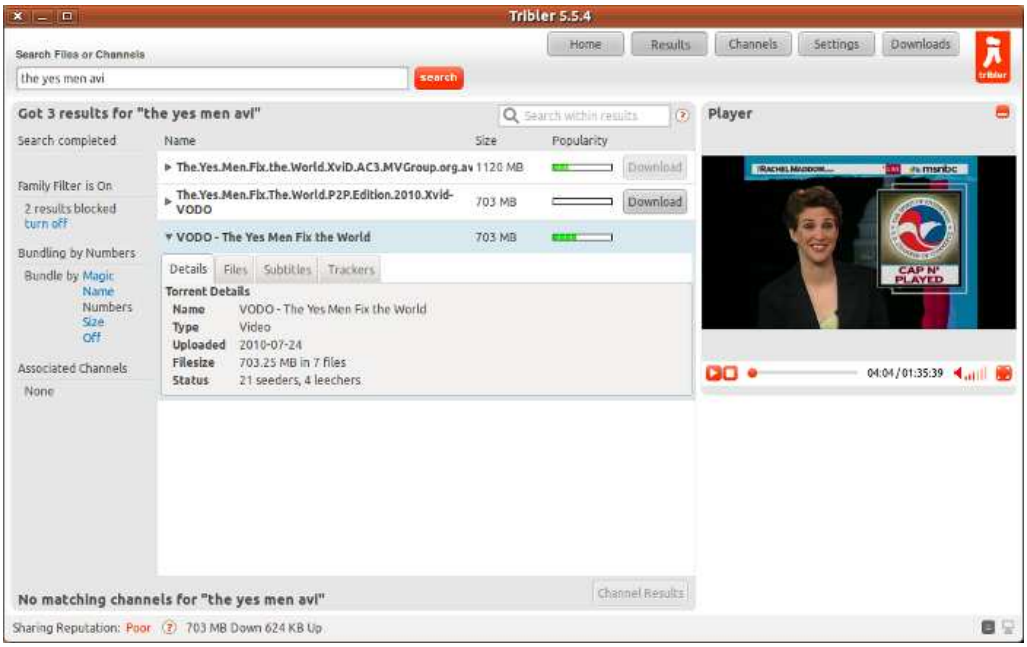

Figure 3.1: The Tribler client in action.

3.1 Implementation overview

Tribler is written in Python2, and is based on the BitTorrent client ABC. It is available for multiple platforms: Windows, Mac, Linux (mainly Ubuntu). The official website of Tribler is `http://www.tribler.org`, and its Subversion repository can be found at `http://svn.tribler.org/`. At this time, the latest major version of Tribler is 5.5.x (see `/abc/branches/release-5.5.x/` in SVN).

The Tribler GUI is created with the help of WxPython, the Python wrapper for the popular widget toolkit WxWidgets. Figure 3.1 shows an example of how Tribler looks in action. The newer versions offer XMLRPC access to the client, compatible with rtorrent. This enables users to have control their client remotely, for example, web interfaces and mobile applications. For local storage, Tribler uses SQLite. This database is mostly made up of data about peers, torrents, and communities. VideoLAN is used as the integrated video player and is accessed via the LibVLC plugin.

Tribler uses its own permission system, called Dispersy. We will discuss Dispersy in detail in the next section.

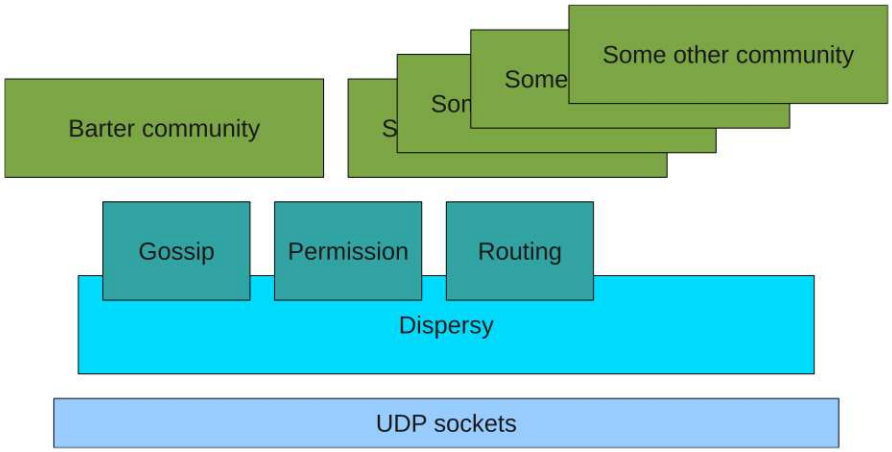

Figure 3.2: A schematic view of Dispersy and other components [17].

3.2 Dispersy

In this chapter we will discuss the Distributed Permission System (Dispersy)[17], which was introduced in the QLectives Platform's version 2.0 as part of Tribler.

Dispersy was introduced in 2011. Technical details can be found in QLectives deliverable D4.1.2[17]. Since then, the system has improved, the deliverable D4.1.3[20] brings version 3.0 of the platform, with many improvements. It also provides results from simulations made on clusters as well as real world deployment.

3.2.1 Main design concepts

Dispersy is designed to be scalable, up to millions of peers. It is designed to have no single point of failure. Security is always hard to provide in a fully distributed environment, and Dispersy only provides mild security solutions. Security is thought to be made strong by social structure. Things are kept simple, so a community designer should only worry about the features for their community.

Each peer initially creates a public-private key pair using Elliptic Curve Cryptography [3]. The SHA1 digest of the public key acts as an identifier for each peer.

Dispersy uses the UDP protocol for transferring data. This simplifies connectability problems because NATs and firewalls are more permissible towards UDP packets than TCP packets.

3.2.2 Communities

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 subsystems. A community shares a set of messages known to all of its members, and a set of permissions implemented by the members. An example community is the Barter Community, introduced in [1].

Community members communicate with messages. Some of these messages are signed by the sender, and propagated throughout the network. There are multiple signing and propagating policies. Within the community, it is possible to send messages to random peers, which the gossip learning algorithms are somewhat dependent on.

Each message has four main attributes, as described in the deliverable:

- Authentication defines whether the message is signed, and if so, by how many members. This defines levels of security through authentication. Possible Authentication policies are: *No-authentication*, *Member-authentication*, and *Multi-member-authentication*.

- Resolution defines how the permission system should resolve conflicts between messages. Permissions can be granted and revoked over time. There are 3 resolution policies: *Public-resolution*, *Linear-resolution*, and *Cyclic-resolution*.

- Distribution defines if the message is sent once or if it should be disseminated among nodes. In the latter case, it can also define how many messages should be kept in the network. The distribution policy can be one of: *Direct-distribution*, *Relay-distribution*, *Sync-distribution*, *Full-sync-distribution*, and *Last-sync-distribution*.

- Destination defines to whom the message should be sent or synchronized. The destination policy defines the target of the message. Possible values are: *Address-destination*, *Member-destination*, *Community-destination*, and *Similarity-destination*.

A community can have multiple types of messages, and for each of those it can define one policy for each attribute. The possible values for these attributes are described in detail in the deliverable.

Each community has a master member, who is the owner of the community. The master member has every permission in the community. The master member's public key is the community ID.

Channels

A special type of community provides channels. These are groups of torrents with attached metadata and comments. People can subscribe to channels, can discuss torrents, vote, mark as spam, and mark as favorite. The users can create their own channels, either by collecting torrents or via an RSS feed. It is possible to make a channel public, so that everyone can edit metadata, add and delete torrents. This is similar to how Wikipedia works, and is a great tool for collaboration. However, people can use this platform to spam and do malicious edits. The edits can be reverted, essentially marking them as spam. This is where we think we can use machine learning for spam filtering and vandalism detection.

Each peer has a local view of channels they subscribe to and it is not feasible to subscribe to all channels. We would like to build models that are available for every peer and captures the global structure of the network so they can use that for filtering spam comments, for example.

Chapter 4

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 a few basic learning algorithms that work on top of the core.

4.1 General overview of the GossipLearningFramework community

We have used the 5.5.x version of Tribler, which was the stable release at the time of this work. 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 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.

We have to be very careful when unserializing data that comes over the network as it

Listing 4.1: 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,
  self.on_receive_model)
```

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, a community can govern 4 different policies of each message: Authentication, Resolution, Distribution, Destination. Our `modeldata` message defines these as shown in Listing 4.1. 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 passive thread callback method is defined to be `on_receive_model`, and the `check_model` method carries out sanity checks on the incoming data. The payload is a `GossipMessage` object which is converted using the JSON format.

The most important parts of the Gossip Learning core, namely the active and passive threads and their helper functions can be seen in Listing 4.2. These functions have the same semantics as the ones in the Gossip Learning Framework described in Section 2.2 of Chapter 2. 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 or P2Pegasos.

4.2 Implementing a learning algorithm

To create a specific learning algorithm, one only has to create a subclass of `GossipMessage`, implementing only `__init__`, `update`, `predict` and optionally `merge`. These functions are also completely analogous with the Gossip Learning Framework described in Section 2.2 of Chapter 2.

Listings 4.3 and 4.4 show the actual source code of the regularized Adaline per-

Listing 4.2: The code of the active and passive threads.

```
def active_thread(self):
  while True:
    self.send_messages([self._model_queue[-1]])
    yield DELAY

def on_receive_model(self, messages):
  for message in messages:
    msg = message.payload.message

    assert isinstance(msg, GossipMessage)

    if self._x == None or self._y == None:
      continue

    self._model_queue.append(self.create_model_mu(msg, self.
       _model_queue[-1]))

def update(self, model):
  for x, y in zip(self._x, self._y):
    model.update(x, y)

def create_model_rw(self, m1, m2):
  self.update(m1)
  return m1

def create_model_mu(self, m1, m2):
  m1.merge(m2)
  self.update(m1)
  return m1

def create_model_um(self, m1, m2):
  self.update(m1)
  self.update(m2)
  m1.merge(m2)
  return m1

def predict(self, x):
  return self._model_queue[-1].predict(x)
```

ceptron and P2Pegasos, respectively. As previously stated, they are both sublcasses of `GossipLearningModel` and they implement the needed functions.

The current implementation of this community offers 3 basic models: Adaline perceptron, Logistic regression, and P2Pegasos. It has a model queue implementation, which enables merging and voted prediction. Throughout the code base, a simple sparse vector implementation is used (using the *dict* structure), the bias term ($x_0 = 1$) is automatically added to the data. This could be improved to use libraries such as `numpy`.

The full source code of the implementation including the evaluation scripts is available at the following git repository:

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

Listing 4.3: The Adaline perceptron model implementation code.

```
class AdalinePerceptronModel(GossipLearningModel):

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

    # Initial model
    self.age = 0

  def update(self, x, y):
    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

    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([self.w[i] * x[i] for i in range(len(self.w))])

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

  def merge(self, model):
      self.age = max(self.age, model.age)
      self.w = [(self.w[i] + model.w[i]) / 2.0 for i in range(
         len(self.w))]
```

Listing 4.4: The P2Pegasos model implementation code.

```
class P2PegasosModel(GossipLearningModel):
  def __init__(self):
    super(P2PegasosModel, self).__init__()

    # Initial model
    self.age = 0

  def update(self, x, y):
    label = -1.0 if y == 0 else 1.0

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

    is_sv = label * sum([self.w[i] * x[i] for i in range(len(
       self.w))]) < 1.0
    max_dim = max(len(self.w), len(x))
    for i in range(max_dim):
      if is_sv:
        self.w[i] = (1.0 - 1.0 / self.age) * self.w[i] + rate *
           label * x[i]
      else:
        self.w[i] = (1.0 - 1.0 / self.age) * self.w[i]

  def predict(self, x):
    inner_product = sum([self.w[i] * x[i] for i in range(len(
       self.w))])
    return 1.0 if inner_product > 0.0 else 0.0

  def merge(self, model):
    self.age = max(self.age, model.age)
    self.w = [(self.w[i] + model.w[i]) / 2.0 for i in range(len(
       self.w))]
```

Chapter 5

Experiments

To assess the correctness and performance of the implementation, we compare our experimental results with the Peersim [15] simulations found in [14].

5.1 Experimental setup

First, we tested the Adaline perceptron, Logistic regression, and P2Pegasos algorithms on two databases, the *Iris* (setosa-versicolor) and *SpamBase* databases [9]. Iris contains 90 training examples and 10 testing examples, while SpamBase contains 4142 training examples and 461 testing examples. The training examples were spread amongst all the peers so that in the case of Iris, each peer has one training example locally. In the case of SpamBase, due to hardware limitations, we used 400 peers with each having 10-11 local training examples.

Tribler uses public-key cryptography with elliptic curves[3] 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 public/private key-pairs for each peer (`Tribler/Core/dispersy/genkeys.py`).

Our experimental scripts started the 90 (400) 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`). In this way, we could do our experiments without starting the Tribler GUI.

Each peer's script is redirecting the standard output and the standard error channels

into logfiles. They also periodically (every 10 seconds) log the timestamp and the model prediction error over the whole testing dataset as well as the number of messages received and the model parameters. The error function we used was the average 0-1 error, which means averaging the ratio of incorrect predictions over the whole network. These data are aggregated (`result.py`) and then plotted (`plot.sh`). In the community, message delay was set to 1 second and peers were started at the same time. We are reproducing the *no failure* and *no churn* scenario.

Peersim[15] is a scalable discrete simulator for peer-to-peer networks written in Java. It supports various overlay networks (e.g. NewsCast) and is highly extendable. The Gossip Learning Framework is implemented on top of Peersim. For the Peersim simulations, we used the similar parameters as in [14] and we only consider the no failure case and no churn.

5.2 Results

Figure 5.1 shows the experimental as well as simulation results for the three models, that is, the maximum 0-1 prediction error over every node over the whole testing set of Iris. Figure 5.2 shows the results for the SpamBase database. In the case of the Iris database, we can see that all three algorithms converge to an error of 0 in every single peer. For Adaline perceptron it takes about 500 seconds, whereas Logistic regression needs about 190 seconds, and P2Pegasos only 110 seconds, which gives a sense of how well these algorithms perform on this dataset.

Since the cycle length was 1 second, the two results should line up. When we compare the number of cycles to the results of the Peersim simulations, we can see that the results are more or less the same, which validates the implementation. The SpamBase database takes longer to learn on, and again, the results are in accordance with results from [14]. Note that the choice of CREATEMODELMU over CREATEMODELUM does not give a huge advantage, however, merging gives a huge edge over simple random walk.

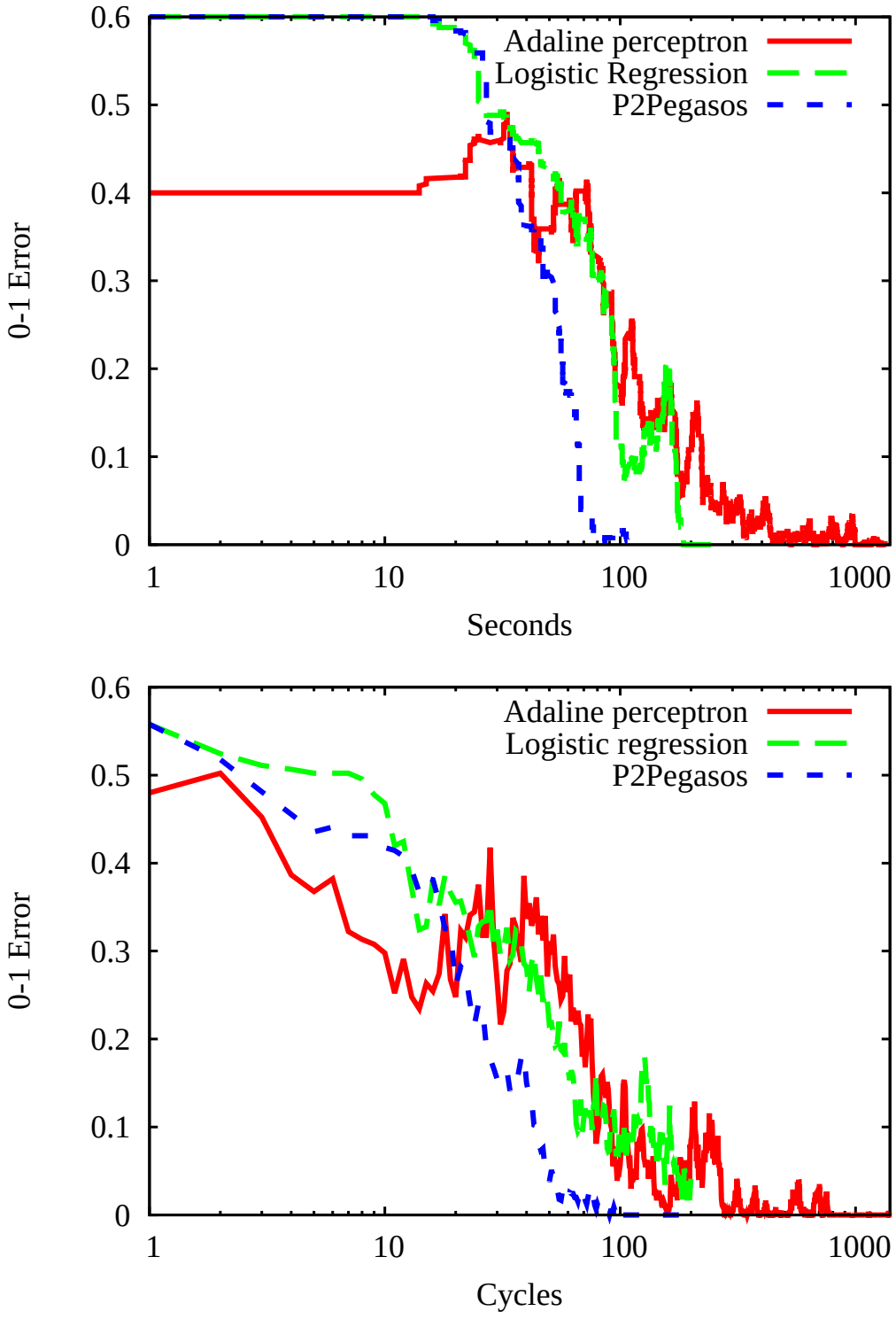

Figure 5.1: Experimental (top) and Peersim simulation (bottom) results for the three algorithms without merge on the Iris database.

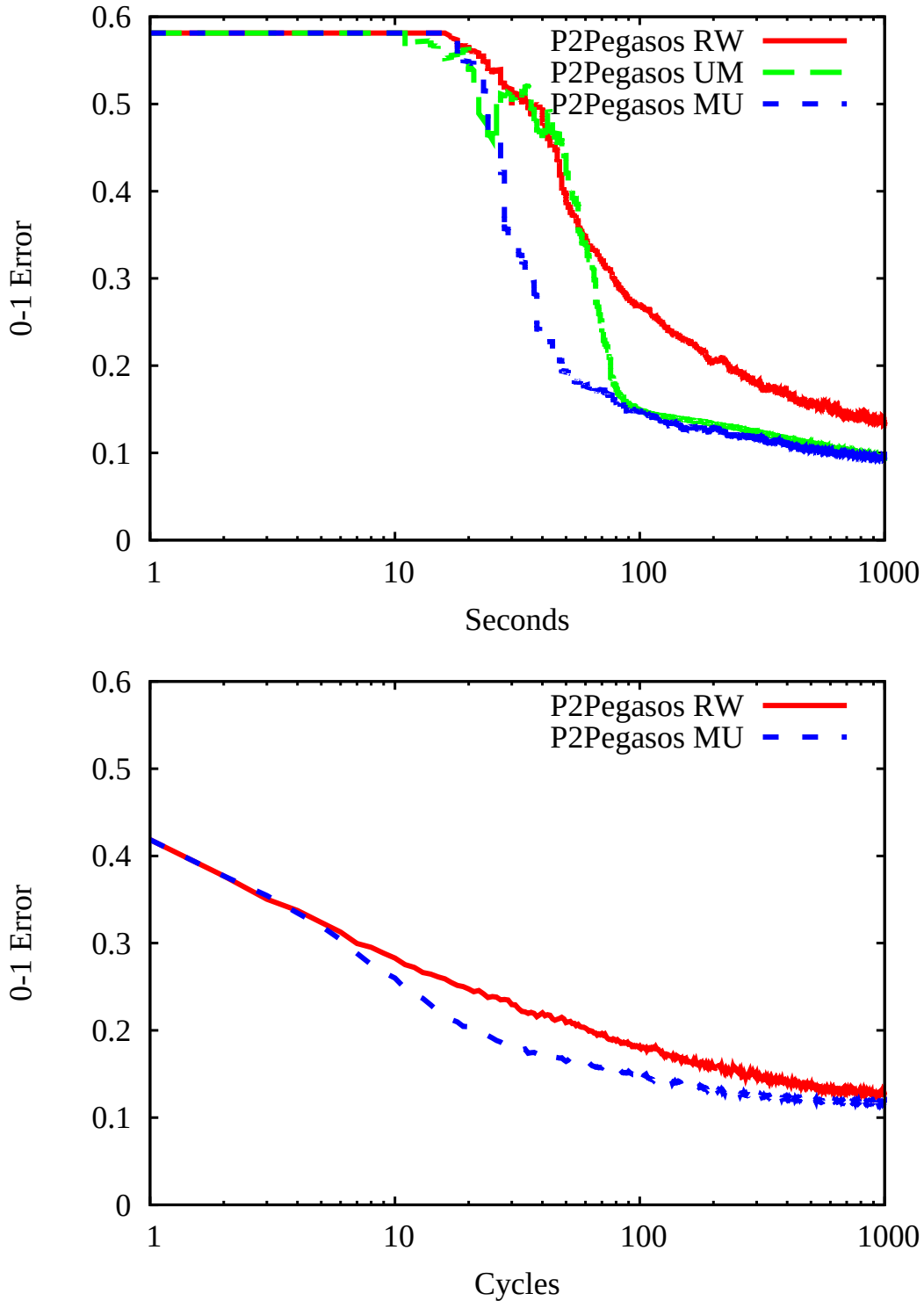

Figure 5.2: Experimental (top) and Peersim simulation (bottom) results for the three algorithms with some merge strategy on the SpamBase database. The simulation framework only contained one type of merge strategy at the time of writing.

Conclusions

We have successfully applied machine learning techniques in an existing distributed system where data is fully distributed amongst peers. The solution provides some privacy, as the local data never leaves the nodes, only the models are transferred. Our experiments show that the implementation performs well in the popular Tribler social peer-to-peer content sharing platform, as a separate community, which is basically a protocol layer.

The process of the implementation has been a great learning experience for all of us and we look forward to cooperating with the developers of Tribler at the Delft University of Technology. Future work includes feeding locally available data in each Tribler client to our protocols. With that, we will be able to learn models globally over the network, even if when the network size grows to millions of users.

Having obtained those models, we can make predictions, which could be particularly helpful for spam filtering or vandalism detection applications as well as any online learning algorithm that can be implemented in GoLF. Having the forementioned services in a network can greatly improve user experience, especially when it is hard to maintain the network due to its size and distributed nature.

Declaration

I, Kornél Csernai, Software Information Technologist MSc. student, declare that this thesis is my work, made at the Faculty of Science and Informatics at the University of Szeged for the purpose of obtaining the degree of MSc. in Software Information Technologist.

I declare that I have not defended this thesis prior, and that it is the product of my own work, and that I only used the cited resources (literature, resources, etc.).

I understand that the University of Szeged places this thesis and makes it publicly available at the Institute of Informatics Library.

Szeged, May 18, 2012 .....

signature

Acknowledgements

I am grateful to Márk Jelasity for his continuous support and direction as my advisor. I am also glad to have been able to discuss machine learning and peer-to-peer problems and the ideas for the integration with Róbert Ormándi and István Hegedűs.

Johan Pouwelse, Niels Zeilemaker, Boudewijn Schoon from the Delft University of Technology played a key role in setting up the initial Tribler community.

I am thankful for Márk Jelasity, Róbert Ormándi, István Hegedűs, Tamás Vinkó, and Veronika Vincze for taking their time and advising me while writing this thesis.

Bibliography

[1] Nazareno Andrade, Tamás Vinkó, and Johan Pouwelse. Qmedia v2 - short report. Deliverable D.4.3.2, QLectives Project, 2011.

[2] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition, 2007.

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

[4] Bram Cohen. Incentives build robustness in bittorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, 2003.

[5] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, September 1995.

[6] L. D'Acunto, J. A. Pouwelse, and H. J. Sips. A measurement of NAT and firewall characteristics in peer-to-peer systems. In Lex Wolters Theo Gevers, Herbert Bos, editor, *Proc. 15-th ASCI Conference*, pages 1–5, P.O. Box 5031, 2600 GA Delft, The Netherlands, June 2009. Advanced School for Computing and Imaging (ASCI).

[7] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2 edition, November 2001.

[8] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 13–13, Berkeley, CA, USA, 2005. USENIX Association.

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

[10] Wojtek Kowalczyk and Nikos Vlassis. Newscast EM. In Lawrence K. Saul, Yair Weiss, and Léon Bottou, editors, *17th Advances in Neural Information Processing Systems (NIPS)*, pages 713–720, Cambridge, MA, 2005. MIT Press.

[11] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1st edition, 1997.

[12] J. J. D. Mol, J. A. Pouwelse, D. H. J. Epema, and H. J. Sips. Free-riding, fairness, and firewalls in P2P file-sharing. In *Proc. 8th IEEE International Conference on Peer-to-Peer Computing*, pages 301–310. IEEE CS, sep 2008.

[13] Andrew Ng. CS299 at Stanford, http://cs229.stanford.edu/.

[14] Róbert Ormándi, István Hegedűs, and Márk Jelasity. Gossip learning with linear models on fully distributed data. *Concurrency and Computation: Practice and Experience*, 2012. to appear.

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

[16] Sandvine. Global internet phenomena report. Technical report, Sandvine, 2011.

[17] Boudewijn Schoon, Tamás Vinkó, and Johan Pouwelse. Qlectives platform v3. Deliverable D.4.1.3, QLectives Project, 2012.

[18] Shai S. Shwartz, Yoram Singer, and Nathan Srebro. Pegasos: Primal Estimated sub-GrAdient SOlver for SVM. In *Proceedings of the 24th international conference on Machine learning*, ICML '07, pages 807–814, New York, NY, USA, 2007. ACM.

[19] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 189–202, New York, NY, USA, 2006. ACM.

[20] Tamás Vinkó, Nazareno Andrade, Boudewijn Schoon, and Johan Pouwelse. Qlectives platform v2. Deliverable D.4.1.2, QLectives Project, 2011.

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