Parallel Natural Language Processing Algorithms in Scala

Stanislav Peshterliev

EPFL

February 4, 2012

Contents

Chapter 1

Introduction

Implementing machine learning algorithms for Natural Language Processing (NLP) on large data sets is hard. Even though much research has focused on making sequential algorithms more scalable, their running times is limited as the data is growing much faster then the computational power of single processor. Meanwhile, parallelization remains challenging for this class of problems.

The aim of this semester project is to implement and benchmark two strategies for parallelizing Maximum Entropy [2] and Naive Bayesian [12] - two widely used machine learning classification algorithms. The project focus is on scalability of the parallel implementation rather then on classification accuracy. We use text categorization as classification task because it is relatively easy to find a large data sets to experiment, and because this is one of the most widely spread and useful Natural Language Processing application, i.e. spam classification.

The parallelization is based on Menthor [7] - a framework for implementing parallel and distributed machine learning algorithms on large graphs developed at EPFL. In terms of Menthor framework architecture, we consider two parallelization strategies: vertex for every sample, and vertex for set of sample; both strategies are analyzed in terms of computational efficiency and scalability, as well as compared to their baseline sequential version.

1.1 Related work

The recent research on parallel machine learning algorithms is focused mainly on implementation based on MapReduce [5] framework, which was proposed by Google. Cheng-Tao Chu et al. have reported a linear improvement in the speed to the number of processes [3] with using map reduce approach. Apache Mahout[1] is an open source project which aim to provide parallel version of the most popular machine learning algorithm. Again, Mahout is using map reduce approach with the open source framework Apache Hadoop[2]

1.2 Overview

The report is structured as follows:

- Chapter 2 presents the Maximum Entropy and Naive Bayes, and their application to text categorization
- Chapter 3 presents the possible parallelization approaches to Maximum Entropy and Naive Bayes, and the two strategies: vertex for every sample and vertex for set of sample
- Chapter 4 discuss the implementation, and the design decisions made
- Chapter 5 presents the experimental design and results from from performance benchmarks and tests

[1] mahout.apache.org
[2] hadoop.apache.org

- Chapter 6 gives conclusions and feature works on the project

Chapter 2

Classification Algorithms

In the field of machine learning classification algorithms fall into the class of supervised learning method. They work by learning from some known set of labeled training samples. The task is to derive a model that given an unlabeled sample determines the class it belongs.

To give context for further discussions, we consider the text categorization problem where samples are documents and classes are categories. The training set for this problem consist of a collection of categorized documents. Classifier have to predict the most likely category of not categorized document. An important step in designing a classifier is deciding which words to take into consideration for the model because there are common words which occur in all classes and using them only obscures the results of classification, for example in English such words are "a", "an", "the". The set of words that we use to represent a document for classification is called feature set, and each word is called feature. In the following section, we will discuss a strategy for automatically determining the most useful feature from training data.

A popular example of text categorization application is spam filtering, where classification algorithm has to label emails or other text documents as spam or not not spam. So thanks to machine learning we receive less junk email which saves us time and trouble.

The following sections give a brief description of Maximum Entropy and Naive Bayes classifiers, and, also, an efficient algorithm for feature selection called Information Gain.

2.1 Maximum entropy

Kamal Nigam et al. describe Maximum Entropy as: A general technique for estimating probability distributions from data. The overriding principle in maximum entropy is that when nothing is known, the distribution should be as uniform as possible, that is, have maximal entropy. Labeled training data is used to derive a set of constraints for the model that characterize the class-specific expectations for the distribution. Constraints are represented as expected values of "features", any real-valued function of an example. The improved iterative scaling algorithm finds the maximum entropy distribution that is consistent with the given constraints. [10]

Let feature be any real valued function on a sample and class, $f : S \times C \to R$. A method for selecting the most descriptive feature for given set of classes is described in section 2.3.

Maximum Entropy distribution has the exponential form:

$$P_\Lambda(c|s) = \frac{1}{Z(s)} \sum_i \exp(\lambda_i f_i(s,c)) \tag{2.1}$$

where λ_i is a parameter that is estimate from the training data for feature i, and $Z(s)$ is a normalizing constant to ensure proper probability.

$$Z(s) = \sum_c \sum_i \exp(\lambda_i f_i(s,c)$$

Note that it is guaranteed that the likelihood surface is convex, having a single global maximum and no local maxima, which means that general optimization technique can be used for estimating parameters Λ [1].

Maximum Entropy classifies sample s as follows:

$$Class(s) = \arg\max_{c \in C} P_\Lambda(c|s)$$

2.1.1 Improved Iterative Scaling

To find the distribution 2.1 with the Maximum Entropy, we use the Improved Iterative Scaling (IIS) [1] algorithm. The training procedure is as follows:

- **Inputs:** A collection S of labeled samples and a set of feature functions f_i.
- For every feature f_i, estimate its expected value on the training samples.
- Initialize all the parameters λ_i's to be zero.
- Iterate until convergence:
  - Calculate the expected class labels for each sample with the current parameters, $P_\Lambda(c|s)$
  - For each parameter λ_i:
    - Set $\frac{\partial B}{\partial \delta_i} = 0$ and solve for δ_i
    - Set $\lambda_i = \lambda_i + \delta_i$
- **Output:** A classifier that takes a unlabeled sample and predicts a class label.

$$\frac{\partial B}{\partial \delta_i} = \sum_{s \in S} (f_i(s|c(s)) - \sum_c P_\Lambda(c|s) f_i(s|c) \exp(\delta_i f^\#(s|c))) \tag{2.2}$$

where $f^\#(s|c) = \sum_i f_i(s|c)$, in case of binary feature this function has the simple interpretation of number of features that are active for a sample.

Equation 2.2 can be solved with a numeric root-finding procedure, such as Newton's method. If $f^\#(s|c) = M$ is constant for all s and c, 2.2 can be solved in closed form as follows:

$$\delta_i = \frac{1}{M} \log \frac{\sum_{s \in S} f_i(s|c(s))}{\sum_{s \in S} \sum_c P_\Lambda(c|s) f_i(s|c)} \tag{2.3}$$

Equation 2.2 is derived from the loglikelihood $l(\Lambda|S)$ for the distribution P_Λ. The loglikelihood is, also, used to evaluate the progress of the training at every iteration. It has the following form:

$$l(\Lambda|S) = \log \prod_{s \in S} P_\Lambda(c(s)|d) = \sum_{s \in S} \sum_i \lambda_i f_i(s, c(s)) - \sum_{s \in S} \log \sum_c exp \sum_i \lambda_i f_i(d, c)$$

2.1.2 Text Classification

To apply Maximum Entropy to text categorization, we need to select a set of words for features. Then for every feature-class combination we instantiate a feature function:

$$f_{w,c'}(s,c) = \begin{cases} 0 & \text{,if c' = c} \\ \frac{tf_{s,w}}{|s|} & \text{,otherwise} \end{cases}$$

where$tf_{s,w}$ is the number of times term w occurs in document s, and $|s|$ is the total number of terms in s. One big advantage of this representation is that we can perform IIS iteration is closed from, which is computationally very efficient [10].

2.1.3 References

More details about Maximum Entropy and Improved Iterative Scaling can be found in [2], [1], [9] and [10]. More details on Maximum Entropy applied to text categorization can be found in Nigam et al. [10].

2.2 Naive Bayesian

Naive Bayesian classifier is based on the Bayes's Rule, which states that:

$$P(C|S) = \frac{P(S|C)P(C)}{P(S)} = \frac{P(S|C)P(C)}{\sum_{c \in C} P(D|C=c)P(C=c)}$$

Bayes's Rule is important because it allows us to write a conditional probability, such as $P(C|S)$ in terms of the "reverse" conditional $P(S|C)$.

Naive Bayes classifies sample s as follows:

$$Class(s) = \arg\max_{c \in C} P(c|s) = \arg\max \frac{P(s|c)P(c)}{\sum_{c \in C} P(s|c)P(c)}$$

Estimating class prior $P(c)$ is straightforward:

$$P(c) = \frac{N_c}{N}$$

where N_c is the number of samples that have class c, and N is the total number of samples.

To estimate $P(s|c)$ we represent the sample s as set of features f_i, and impose the simplifying assumption that f_i is independent from f_j for every $i \neq j$ which is a requirement for Naive Bayes classifier. This means that $P(s|c)$ can be written as:

$$P(s|c) = \prod_{i=1}^{n} P(f_i|c)$$

Note that, feature selection is discussed in section 2.3

2.2.1 Text Classification

To apply Naive Bayes to a text categorization, we need to select a set of words for features. Then for every feature-class combination estimate the probability of $P(s|c)$ from training data.

For our classifier we use Multinominal event space representation in which opposite to multiple-Bernoulli event space representation features are not binary [4]. Thus $P(s|c)$ is estimated as follows:

$$P(w|c) = \frac{tf_{w,c}}{|c|}$$

where $tf_{w,c}$ is the number of times term w occurs in class c in the training set, and $|c|$ is the total number of terms that occur in training set with class c

Given the Multinominal distribution the likelihood of document s given class c is computed according to:

$$P(s|c) = \prod_{w} P(w|c)^{tf_{w,s}}$$

where $tf_{w,d}$ is the number of times that term w occurs in document s.

2.2.2 References

Accessible introduction to Naive Bayes and text categorization can be found in Croft, Metzler and Strohman [4]. More detailed description with experimental results can be found in Rennie, Shih, Teevan and Karger [12].

2.3 Information Gain

For text classification, it is common to have one or more features for every word in the training data. Thus, depending on the problem , feature set can be very large. Since feature set size affects both efficiency and effectiveness of the classifier, we want to select a subset of features such that efficiency is significantly improved. Usually not only the efficiency is improved but also effectiveness because some features are noisy or inaccurate. The process of selecting most useful features is called *feature selection*.

Information gain is one of the most widely used feature selection criterion for text categorization applications. Information gains measures how much information about the class is gained when some feature is observed. For example the feature "cheap" gives more information about the class spam then the feature "the".

Information gain measures how the entropy of $P(c)$ changes after we observe feature f. Thus we compute information gain for f as follows:

$$IG(w) = H(C) - H(C|w) = -\sum_{c \in C} P(c) \log P(c) + \sum_{w \in 0,1} P(w) \sum_{c \in C} P(c|w) log P(c|w)$$

2.3.1 References

Introduction on Information gain can be found in Croft, Metzler and Strohman [4]. Comparison between Information Gain and other feature selection criterion is done by Yang and Pedersen [14].

Chapter 3

Parallelization

3.1 Maximum Entropy

Mann, McDonald, Mohri, Silberman and Walker [8] describe three methods for parallelizing Maximum Entropy model training: Distributed Gradient Method, Majority Vote Method and Mixture Weight Method. We implemented Mixture Weight Method because of its computational efficiency and small communication overhead.

Using Mixture Weight Method the training data is split in p partitions $S_1, S_2, ..., S_p$, and training is performed separately on every partition, on each iteration the parameter vector Λ_i of process i is exchanged with other training processes, then the parameters are mixed as follows:

$$\Lambda_\mu = \sum_{k=1}^{p} \mu \Lambda_k$$

The resulting vector can be used directly for classification. We set μ to be $\frac{1}{p}$, where p is number of processes.

One disadvantage of this method is that if the training set is small and p is large, each parameter vector Λ_i is estimated on small training data which leads to bad models. But when considering large scale machine learning as in our case this is not a problem.

3.2 Naive Bayes

Naive Bayes classifier training consists in estimating the probabilities $P(c)$ and $P(s|c)$ on the training set S. How these probabilities are estimated is dependent on the specific application. In case of text categorization we need: N_c - number of samples that have class c; $tf_{w,s}$ - number of times term w occurs in sample s; $tf_{w,c}$ - number of times term w occurs in class c; and $|c|$ - total number of terms in class C.

In order to parallelize Naive Bayes applied to text categorization we split the training data in p partitions $S_1, S_2, ..., S_p$, where p is number of processes. Then we estimate N_{ci}, tf_{w,s_i}, tf_{w,c_i}, and $|c|_i$ on every partition S_i. Finally, we mix N_{ci}, tf_{w,s_i}, tf_{w,c_i}, and $|c|_i$ to obtain a model trained on S.

3.3 Menthor framework

To parallize the algorithms functionality we use Menthor [7]. Menthor is a framework for parallel graph processing, but it is not limited only to graphs. It is inspired by BSP with functional reduction/aggregation mechanisms. The framework aims to ease parallelizing machine learning functionality and it is is implemented in Scala programming language with the Actor model.

In Menthor all data to be processed is represented as a data graph. Vertices in such a graph typically represent atomic data items, while edges represent relationships between these atoms. In addition to the data item that it represents, a vertex stores a single value that is iteratively updated during processing - thus, an algorithm is implemented

by defining how the value of each vertex changes over time. At each step the algorithm a vertex can exchange and receive messages from other vertices. Thus, the update of a vertex's value is based on its current value, its list of incoming messages, as well as its local state. During the execution of an iteration each update step on different vertices is executed in parallel.

For example, page rank algorithm can be implemented as follows:

```
class PageRankVertex extends Vertex[Double](0.0d) {
  def update() = {
    var sum = incoming.foldLeft(0)(_ + _.value)
    value = (0.15 / numVertices) + 0.85 * sum
    if (superstep < 30) {
      for (nb <- neighbors) yield Message(this, nb, value / neighbors.size)
    } else
      List()
  }
}
```

Listing 3.1: Page rank algorithm

More information about Menthor can be found in the original technical report by Philipp Haller and Heather Miller [7], and in the presentation "The Many Flavors of Parallel Programming in Scala" by Philipp Haller [6].

Thanks to Menthor's ability to mix sequential and parallel programming as a combination of superstep, translating the sequential version of an algorithm to is parallel version straightforward, and it is not discussed in here. The next two section discuss the two parallelization strategies that we implemented on top of Menthor for Maximum Entropy and Naive Bayes classifiers training.

3.3.1 Strategy 1: Vertex for every sample

The data is partitioned into p partitions, and then the samples from every partition are added as vertices in the graph, moreover, for every partition there is a master vertex that is used to aggregate the results generated from samples in the partition, masters are connected in order to be able to exchange messages. See figure 3.1.

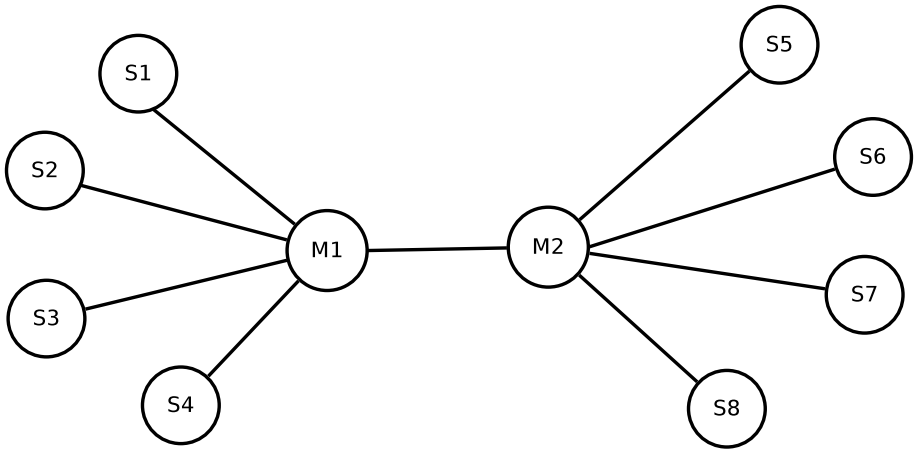

Figure 3.1: Vertex for every sample graph

With this strategy in order to compute some global function over the whole training set, sample vertices have to send $|S|$ messages to masters, and then p master exchange messages between them, such that every master will have the result of the global function. Thus in total $|S| + p(p-1)$ messages are necessary.

The communication overhead of this strategy is significant especially if the vertices have to exchange messages with a lot of data. Also, if the data is partitioned in large number of groups the number of exchanged messages grows fast because of the term $p(p-1)$.

3.3.2 Strategy 2: Vertex for set of samples

The data is partitioned p partitions, but unlike vertex for every sample strategy, only p master vertices are created and added to the graph, each vertex contains the set of samples for the given partition. Thus data on every vertex is processed in parallel. See figure 3.2.

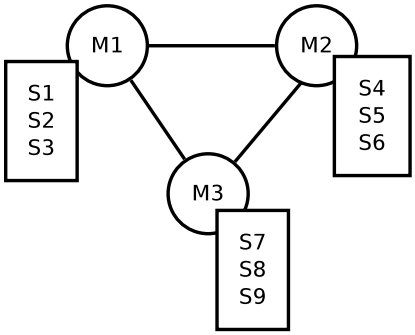

Figure 3.2: Vertex for set of samples

The advantage of this strategy is that only $p(p-1)$ messages are needed in order to compute a global function over this graph. Thus for big $|S|$ and small p this strategy has much less communication overhead compared to vertex for every sample. Also, the strategy uses less memory because we do not create sample vertices .

Chapter 4

Implementation

4.1 Technologies

The project is entirely implemented in Scala programming language[1], except for one method that is implemented in Java[2] that was needed in order to avoid JVM Double autoboxing which significantly impacts numerical operation performance.

The excellent Scalala [3] numeric linear algebra library is used for numeric computation and sparse vectors support. The library also provides rich Matlab-like operators on vector and matrices, as well as on standard Scala data structures like Lists, Sets, Arrays and Maps.

We are, also, using Trove[4] high performance collection library for Java, which is very easy to do from Scala thanks to its seamless integration with Java. We do that because when dealing with a lot of numeric data on JVM it is very important to minimize autoboxing due to the impact on performance, and, also, to save memory. The problem with standard collections is that every time you write a double number to Java or Scala hash map it is automatically box to its Double object, Trove provides optimized collections that avoid this behaviour.

We are using Sbt [5] to manage project dependencies and organize the build procedure.

4.2 Organization

4.2.1 Package structure

- menthor.apps - contains classes related to applying the algorithms to text categorization, and classes for experimenting with different data sets
- menthor.classifier - contains abstract base classes and traits that define the general interfaces for classification algorithms
- menthor.classifier.featureselection - contains implementation of Information Gain feature selection algorithm
- menthor.classifier.maxent - contains implementation of Maximum Entropy classifier
- menthor.classifier.naivebayes - contains implementation of Naive Bayes classifier
- menthor.util - contains utility classes for working with collections, files and probability distributions

[1] www.scala-lang.org
[2] www.java.com
[3] github.com/scalala/Scalala
[4] trove.starlight-systems.com
[5] https://github.com/harrah/xsbt

4.2.2 Class hierarchy

There are two main class hierarchy Classifier on figure 4.1 and Trainer on figure 4.2.

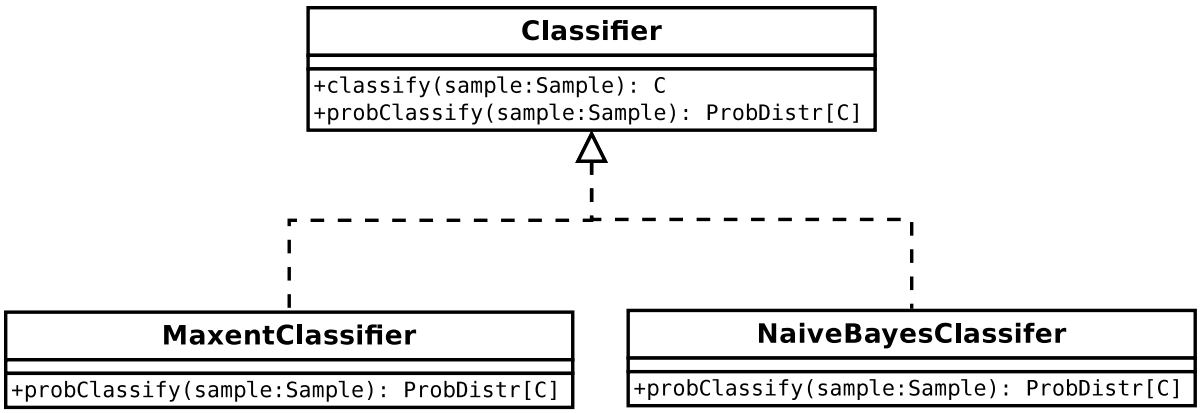

Figure 4.1: Classifier class heiracy

As depicted on figure 4.1 there are two classes that inherit from Classifier abstract class: MaxentClassifier and NaiveBayesClassifier. Classes inherited from Classifier have to implement probClassifiy method that returns the probability distribution for every class of the classifier. The method classify from Classifier uses probClassify and returns the class with highest probability.

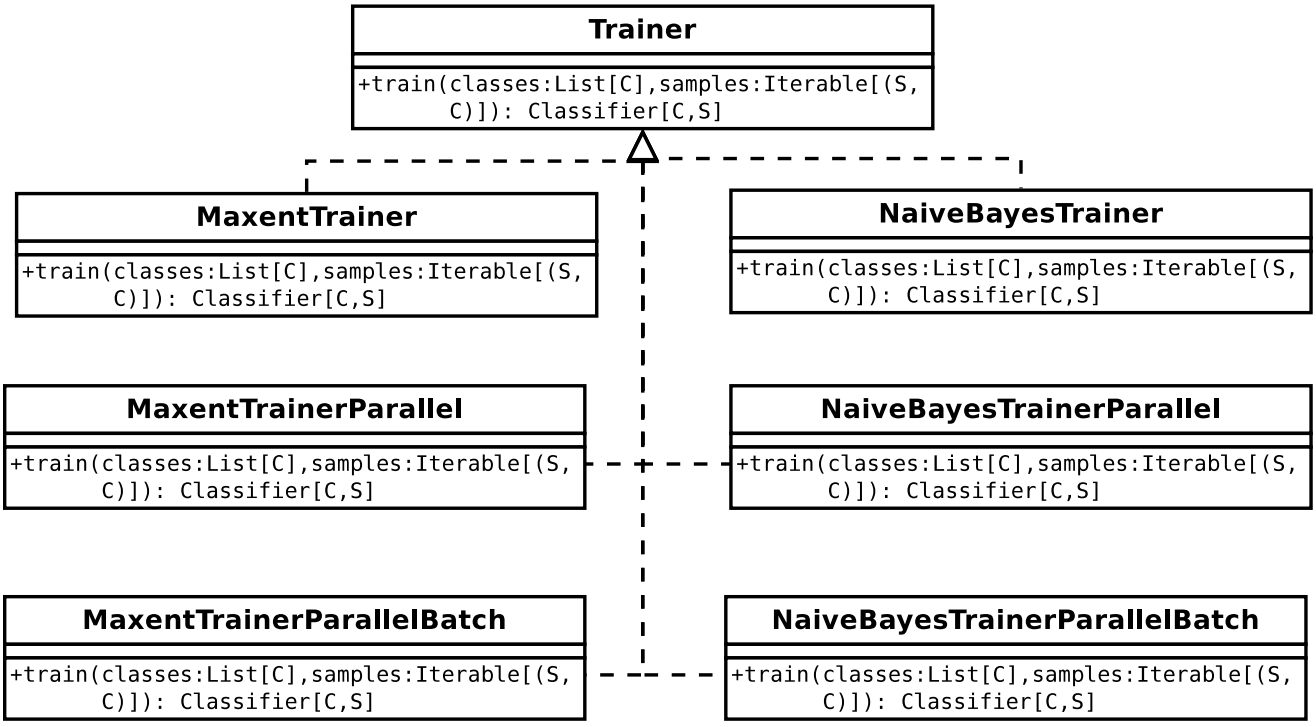

Figure 4.2: Trainer class hierarchy

There are six classes implementing Trainer trait, three for Maximum Entropy and three for Naive Bayes. Classes without any suffix implement sequential version, these with suffix Parallel implement vertex for every sample strategy,

and ParallelBatch implement vertex for set of samples strategy.

4.3 Extensions

New classifiers should extend Classifier abstract class and Trainer trait, and implement the corresponding abstract methods. Different feature selection algorithm can be implemented by extending FeatureSelector trait and implementing ”select” method.

In order to apply the algorithms for different task, the trait Sample have to be implemented and features for the given task have to be represented as a map from integer to double for the given sample.

Chapter 5

Experimental results

5.1 Training data sets

We are experimenting with three data sets: Movie Reviews [11], 20 Newsgroups [1] and Wikipedia INEX 2009 collection [13].

5.1.1 Movie Reviews

Movie Reviews dataset contains 2000 movie reviews: 1000 positive and 1000 negative. The task is to train a classifier which decides if unseen movie review is negative or positive. The data set is small, and uninteresting for large scale machine learning but it was used at the beginning of the development to test the correctness of the algorithms. The classification accuracy for this corpus is given in table 5.1.1.

Algorithm	Accuracy
Maximum Entropy	86.33
Naive Bayes	85.62

Table 5.1: Movie Reviews data set classification accuracy

At each separate run the data is split into training and test sets. The test set is 10% of the the collection and training set is the rest. The test set is determined at random for each run, thus the accuracy from the table is calculate as average of 5 runs of the algorithms.

We have empirically found out that the best accuracy is achieved with 100 features, which means that Maximum Entropy model has 200 parameters.

5.1.2 20 Newsgroups

The data set consists of approximately 25000 messages taken from 20 newsgroups, categorized into 20 categories: alt.atheism, comp.graphics, comp.os.ms-windows.misc, comp.sys.ibm.pc.hardware, comp.sys.mac.hardware, comp.windows.x, misc.forsale, rec.autos, rec.motorcycles, rec.sport.baseball, rec.sport.hockey, sci.crypt, sci.electronics, sci.med, sci.space, soc.religion.christian, talk.politics.guns, talk.politics.mideast, talk.politics.misc, talk.religion.misc.

Unlike Movie Reviews with this data set the training and test sets are predefined as follows: 18846 training examples, and 7532 test examples. The classification accuracy for this data set is given in table 5.1.2.

On this data set the accuracy of Maximum Entropy is worse than Naive Bayes, to ensure that our implementation of Maximum Entropy is correct we compared the results with the results from Apache OpenNLP[2] Maxent package.

[1] people.csail.mit.edu/jrennie/20Newsgroups

[2] incubator.apache.org/opennlp

Algorithm	Accuracy
Maximum Entropy	57.44
Naive Bayes	92.12

Table 5.2: 20 Newsgroups classification accuracy

OpenNLP categorized accuracy with default settings is 52.60% which is even worse then our implementation. We did not optimize further the accuracy of Maximum Entropy to match the results of Naive Bayes because for this project we are mainly interested in scalability and performance.

Because of the large number of classes for this task the classification is performed on a set of 5000 features, thus Maximum Entropy model has 20000 parameters.

5.1.3 Wikipedia INEX 2009 collection

This data set was prepared for evaluating information retrieval tasks but we adopted it to text classification because the wikitext is conveniently cleansed and parsed to XML which is standard and easy to use. It consist of 2,666,190 articles from Wikipedia. In our experiment we use a subsets of 40,000 articles.

One of the issues with using this data set for text categorization is that categories are not well defined and sparse, such as: "Person who immigrated to Canada" or "Persons who won the noble price", we would like to have both articles in "person" category. To achieve this we use the 331 classes defined in Dbpeida[3] ontology, and try to do a keyword matching on article category to dbpedia class in order to determine to which dbpedia class the article belongs.

We use a subset of 10000 features for this data sets. We did not test for accuracy on this data set, and it is only used for performance benchmarks.

5.2 Data set preprocessing

The data sets that are used comes in plain text or XML. Parsing large number of files is time consuming and does not give information regarding the performance of the algorithms that we test, and only may makes it harder to do accurate performance analysis. Because the algorithms are run many time against the same data minimizing the parsing time and I/O time is important, that is why data data is parsed and compressed in advanced before doing benchmarking. The preprocessing procedure is as follows:

First, all training data is loaded and feature selection is performed, after the most useful features are decided each features is given a unique id, and the triple id, features, and information gain is written to a files. For example see listing 5.1.

```
2507 | bad | -0.2579069473576291
3081 | worst | -0.2638446978715786
1425 | stupid | -0.2752135575766913
8286 | boring | -0.27746205051618666
1110 | waste | -0.2813770454287048
.......
```

Listing 5.1: Features file extract

Second, the data set compression is performed; only selected features are retained and the samples are written in single file with format: category | sample name | feature_id:feature_value feature_id:feature_value. For example see see listing 5.2.

```
pos | cv889_21430.txt | 282:1.0 280:1.0 275:1.0 272:1.0 260:1.0 259:1.0 258:1.0 256:2.0
pos | cv477_22479.txt | 99:2.0 195:1.0 273:1.0 455:1.0 391:1.0 380:2.0 376:2.0 178:1.0
```

[3] http://dbpedia.org/About

```
neg | cv314_16095.txt | 757:1.0 2163:2.0 3696:1.0 5738:2.0 7505:1.0 8286:1.0 3181:1.0
neg | cv690_5425.txt | 367:4.0 1557:2.0 2102:4.0 2814:2.0 1586:2.0 23901:2.0 8:7.0
.......
```

Listing 5.2: Preprocessed samples

5.3 Command line utilities

The project has the following command line utilities for working with the data sets:

- Given a data set this command line program selects features and creates preprocessed training file. The [data set] parameter can be moviereviews, newsgroups or wikipedia.

```
parallelnlp menthor.apps.BuildCorpus [data set] [features size] [data set path] [output
    training file] [output feature file]
```

- Performs classification on movie reviews data set. [algorithm] parameter: can be maxent or naivebayes; and [training mode]: sequential, parallel, parallelbatch.

```
parallelnlp menthor.apps.MovieReviewClassifier [algorithm] [traning mode] [movie reviews
    training file] [features file]
```

- Performs classification on 20 Newsgroups data set. [algorithm] parameter: can be maxent or naivebayes; [training mode]: sequential, parallel, parallelbatch; and [evaluation] can be true or false. Turning off evaluation is useful for benchmarking when you what to only know the performance of the parallel part.

```
parallelnlp menthor.apps.NewsgroupsClassifier [algorithm] [traning mode] [newsgroups training
    file] [newsgroups test path] [features file] [evaluation] [benchmark output file]
    [benchmark iterations]
```

- The same options as for 20 Newsgroups, but works on wikipedia corpus.

```
parallelnlp menthor.apps.WikiepdiaClassifier [algorithm] [traning mode] [newsgroups training
    path] [newsgroups test path] [features file] [evaluation] [benchmark output file]
    [benchmark iterations]
```

5.4 Performance Benchmarks

Algorithms are tests with 1,2,4,6,8 cores, for each core we have performed 5 iterations but we did not take consider the first one because JVM JIT compiler still did not optimize the code at this stage and it usually has lower performance.

5.4.1 Maximum Entropy

- Vertex for a set of sample

  In table 5.4.1 and figure 5.1 you can see that there is an average improvement of 1.65x times for every 2 cores. The biggest improvement is between 1 and 2 cores - 1.86x times. The overall improvement between 1 and 8 cores is 7.38x times.

Cores	Average Time
1	346361.75
2	185633.75
4	111657.75
6	73359.75
8	46883.5

Table 5.3: Maximum Entropy results

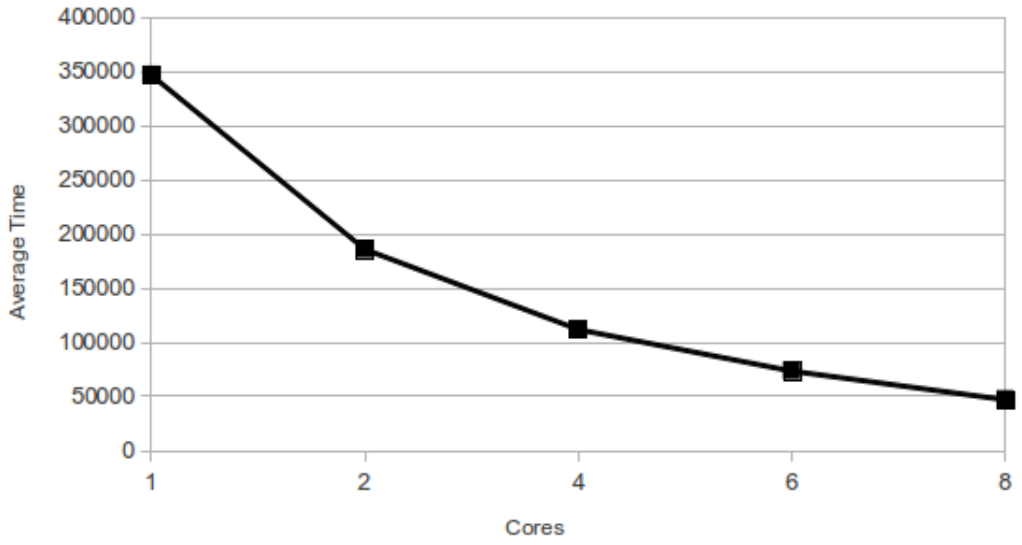

Figure 5.1: Maximum Entropy plot

- Vertex for every sample

  The average time for this strategy on 8 cores is 5241952 ms which around 111x times worse then the time for vertex for set of samples strategy. This results shows that vertex for every sample can be impractical for some tasks, especially when the data is not naturally a graph and can be processed in other manner.

- Sequential

Cores	Average Time
1	67235
8	67140.75

Table 5.4: Maximum Entropy results

As expected there is no speed up between 1 and 8 cores for the sequential version. The comparison with 8 cores of parallel version shows that parallel version is only 1.40x times then the sequential version, which means that due to parallelization overhead the parallelized algorithms should be applied only on large datasets, and on machines with many processors in order for the communication overhead to be paid off.

5.4.2 Naive Bayes

- Vertex for a set of sample

Cores	Average Time
1	8509
2	4731.5
4	3071.5
6	2247.25
8	1619.75

Table 5.5: Naive Bayes results

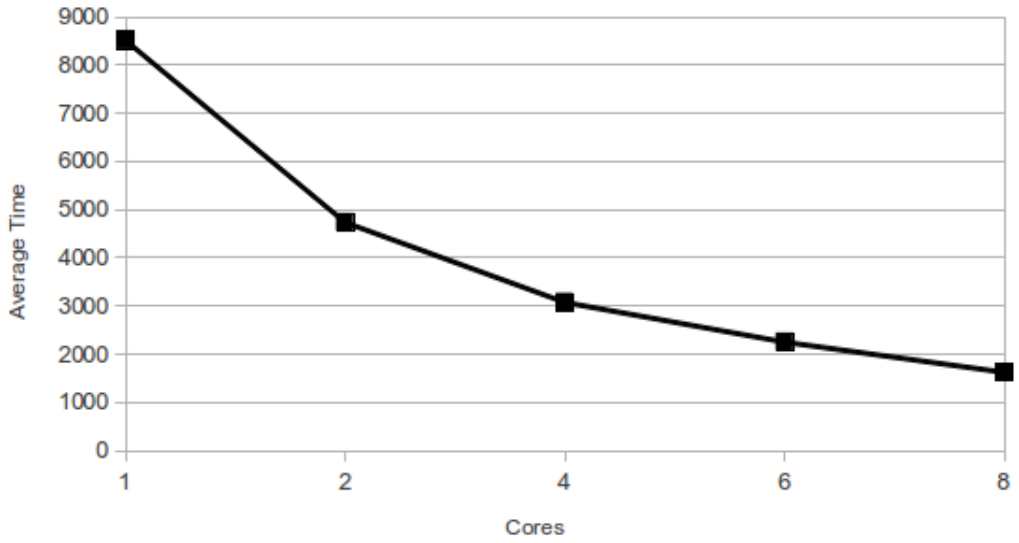

Figure 5.2: Naive Bayes plot

In table 5.4.2 and figure 5.2 you can see that there is an average improvement of 1.52x times for every 2 cores. The biggest improvement is ,again, between 1 and 2 cores - 1.79x times. The overall improvement between 1 and 8 cores is 5.25x times. The overall improvement is smaller compared to Maximum Entropy because Naive Bayes is computationally much less expensive.

- Vertex for every sample

  The average time for this strategy on 8 cores is 62300.5 ms which around 38x times worse then the time for vertex for set of samples strategy.

- Sequential

Cores	Average Time
1	5224.75
8	5579.75

Table 5.6: Naive Bayes results

As expected there is no improvement in the running time between 1 and 8 cores for the sequential version. The comparison with 8 cores of parallel version shows that parallel version is only 3.10x times then the sequential version, which is better than the improvement with Maximum Entropy.

Chapter 6

Conclusions and Feature work

Panellizing machine learning algorithms is challenging but frameworks like Menthor makes it much easier. With this project we have shown that we can easily implement a parallel version of algorithms without much effort and with good scalability.

It will be interesting to implement other popular machine learning algorithms like SVM and Neural Networks. Also, due to Menthor distributed computation features the algorithms can be run transparently on clusters of computers, the results from such experiments would give us some insight on how communication overhead in a network of computers affects the performance, and data sets of what size are appropriate for distributed computation.

Bibliography

[1] Adam Berger. The improved iterative scaling algorithm: A gentle introduction. January 13 1997.

[2] Adam L. Berger, Stephen A. Della Pietra, and Vincent J. Della Pietra. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71, 1996.

[3] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.

[4] W. Bruce Croft, Donald Metzler, and Trevor Strohman. *Search engines: information retrieval in practice*. Pearson Education, Boston, MA, USA, 2010.

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

[6] Philipp Haller. The many flavors of parallel programming in scala. Presented at Scalathon 2011 at the University of Pennsylvania in Philadelphia, 2011.

[7] Philipp Haller and Heather Miller. Parallelizing machine learning- functionally: A framework and abstractions for parallel graph processing, 2011.

[8] Gideon Mann, Ryan T. McDonald, Mehryar Mohri, Nathan Silberman, and Dan Walker. Efficient large-scale distributed training of conditional maximum entropy models. In Yoshua Bengio, Dale Schuurmans, John D. Lafferty, Christopher K. I. Williams, and Aron Culotta, editors, *NIPS*, pages 1231–1239. Curran Associates, Inc, 2009.

[9] Christopher Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, 1999.

[10] Kamal Nigam, John Lafferty, and Andrew Mccallum. Using maximum entropy for text classification, April 29 1999.

[11] Bo Pang and Lillian Lee. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In *Proceedings of the ACL*, 2004.

[12] Jason Rennie, Lawrence Shih, Jaime Teevan, and David Karger. Tackling the poor assumptions of naive bayes text classifiers. In *Proceedings of ICML-03, 20th International Conference on Machine Learning*, Washington, DC, 2003. Morgan Kaufmann Publishers, San Francisco, US.

[13] Ralf Schenkel, Fabian M. Suchanek, and Gjergji Kasneci. YAWN: A semantically annotated wikipedia XML corpus. In Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, and Christoph Brochhaus, editors, *BTW*, volume 103 of *LNI*, pages 277–291. GI, 2007.

[14] Yiming Yang and Jan O. Pedersen. A comparative study on feature selection in text categorization. In *International Conference on Machine Learning*, pages 412–420, 1997.