PARALLEL SUPPORT VECTOR MACHINES USING SMO

by

NAMAN FATEHPURIA

(Under the Direction of John A. Miller)

ABSTRACT

We present an algorithm for Support Vector Machines that can be parallelized effectively. The Algorithm scales up nicely on very large datasets of million training points. Instead of analyzing and optimizing the whole training set in to one support vector machine, the data is split into subsets and each subset is optimized independently on different Support Vector Machine. The result from each Support Vector Machine are then combined to get the trained Support Vector Machine. The high performance is due to low overhead communication between the different Support Vector Machines. In this paper, the runtime performance of the algorithm is tested on a dataset of more than 8 million instances with a speed up of about 20 fold.

INDEX WORDS: Parallel support vector machine, Sequential minimal optimization

PARALLEL SUPPORT VECTOR MACHINES USING SMO

by

NAMAN FATEHPURIA

B.Tech, Punjab Technical University, India, 2010

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2014

PARALLEL SUPPORT VECTOR MACHINES USING SMO

by

NAMAN FATEHPURIA

Major Professor:	John A. Miller
Committee:	Lakshmish Ramaswamy Krzysztof J. Kochut

Electronic Version Approved:

Julie Coffield
Interim Dean of the Graduate School
The University of Georgia
December 2014

DEDICATION

To my friends, family and professors for their endless support, care, belief and motivation.

ACKNOWLEDGEMENTS

Studying in University of Georgia was a great experience for me from past 2 years. I have grown as a better person on both personal and professional front. The knowledge that I gained here is incredible. I learned a lot from my major professor Dr. John A. Miller. He is one of the most intellectual person I have ever met. His knowledge and support has always been the motivating factor for me in completing my research successfully. Also, I would like to extend my gratitude to Dr. Lakshmish Ramaswamy for his endless support and advice. Dr. Krzysztof J. Kochut is one of the best professors and I really enjoyed his Enterprise Integration class.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION AND LITERATURE SURVEY

"Machine learning deals with constructing algorithms and study of system that can learn from data, rather than follow only explicitly programmed instructions" [26]. A key task performed by machine learning systems is to learn from a function that maps input data to the certain outputs.

Machine learning tasks can be categorized into many types like supervised learning and unsupervised learning. In supervised learning, a machine utilizes the training data to train itself. Training data is generally expressed in the form (X, y), where X is referred as data instance and y is referred as the output of X. The goal of supervised machine learning is to predict the value of test data which are expressed in terms of X and the predicted value in terms of y.

Two most common supervised learning tasks are classification and prediction. In classification, output categories or classes can only be of finite number but in prediction, the output set can be of real numbers which is infinite.

In unsupervised learning, the most common example is that of data clustering. The goal of data clustering is to construct a function that partitions datasets into say k different clusters. Data instances that are assigned to the same cluster should have some similarity like having a common centroid or are certain distance apart.

1.1.Problems: dealing with machine learning

The potential training data for machine learning can be extremely large. The number of data instances can reach up to million, where each data set can have thousands of non-zero features. Storage required for this humongous data can be an issue while dealing with machine learning.

Features dimensionality can also prove to be a problem in machine learning - in some cases like images and video, input dimensionality can be in million or more. Computations on these many features may prove to be challenging.

In machine learning, algorithm can be very complex which increases the computational complexity of the complete model. The learning problem can be too slow for such complex model. For these kinds of problems, parallel multi-node or multi-core implementation can be a solution.

Throughput and latency can also be an issue in machine learning. Many machine learning problem require performing a sequence of interdependent tasks, which is viewed as a single task, which typically results in very high computational costs due to increased computational complexity.

1.2. Parallel and distributed machine learning

Performance can be gained in machine learning applications by the concurrent execution of tasks that may be otherwise performed sequentially. Two major areas in which concurrency can be obtained are parallel machine learning and distributed machine learning.

1.3.Pros and cons: parallel and distributed machine learning

1.3.1. Advantages: Parallel Machine Learning

Parallel implementation of machine learning uses shared memory as an underline concept. Algorithms implemented using shared memory can make use of shared global data during concurrent execution. System threads and program threads can make use of the shared state which will result in faster execution of the program.

Parallel implementation can be less resource intensive because we can process multiple client requests concurrently.

Parallel implementation has the ability for an application to remain responsive to input because in case of single threaded application, if the main execution thread blocks for a very long task, the entire application can be blocked.

High levels of parallelism can be achieved by utilizing multi-core systems. Each core may have two system threads, so more the number of cores in the system, more will the number of system threads in the system. We can split the data into much smaller chucks because of the increased system threads and run them in parallel, which will result in a faster execution time.

Parallel implementations can be cost effective, have better system utilization, have low network communication overhead and generally be easy to code.

1.3.2. Disadvantages: Parallel Machine Learning

The primary disadvantage of parallel implementations is the lack of scalability both in terms of memory and number of cores in the system. For a very large dataset say in the billion, it is very hard to keep all the data and process the data on the one machine

because the memory required for this is huge. Adding more cores can increase the traffic between the shared memory and CPU path.

Another major disadvantage of parallel processing is the program ending up in race conditions or deadlock conditions. Since threads share the same address space, the program needs to handle the race conditions. In order for data to be correctly manipulated, threads need to process the data in the correct order. To maintain the order, threads needs to use some kind of locking mechanism in order to prevent common data from simultaneously being modified or read while in the process of modification.

1.3.3. Advantages: Distributed Machine Learning

A distributed system is a collection of many independent computer systems, each having its on local memory and processor. We can divide the large dataset on to these different machines and can process them; this will eliminate the memory and processing problem faced in parallel implementation.

Distributed systems are scalable, with the increase in data or processing needs, we can add more computers to the system.

Distributed systems are reliable, as data is replicated on different machines, so even in case of machine failure, data can be reconstructed and processing can be resumed without making the complete system at halt.

A distributed system offers a better price and performance then single system because it is cheaper to get many computers with fewer cores than a single computer with many cores.

1.3.4. Disadvantages: Distributed Machine Learning

In distributed systems, troubleshooting and diagnosing problem can become more difficult, because the analysis of the system may require connecting to remote nodes or inspecting communication between nodes.

Distributed systems performance depends highly on bandwidth consumed and latency. If these are too significant then the benefits of distributed computing may be negated and the performance may be worse than a non-distributed environment. Security can also be an issue with distributed system.

1.4. Techniques: parallel processing

1.4.1. Data Splitting

1.4.1.1. Instance Shards

When we split the data for the machine learning technique in terms of the instances then each set of instances is known as instance shards.

Data instances for each shard can be selected either randomly or sequentially, or can be selected in terms of certain values in the instances. Once the shards of different data instances are created, each shard is assigned to a thread, so that each shard can be run in parallel.

After running each shard on different thread, we still need a way to combine the result back from each thread to build a model for the machine learning algorithm.

1.4.1.2. Feature Shards

In feature shard, we split the data for learning algorithm in terms of number of features. Each shard contains different features for each data instance. Each thread will run each shard independently and each shard will have all the data instances but only few features for each data instance. A result for each data instance is calculated by combining the results from each shard for that particular instance. Results can be combined by simple addition or by treating the result from each shard as a new feature and then running the newly created model again.

1.4.2. Feature Selection

Features in a machine learning algorithm play very important role in creating a well build model. Adding extra features with unnecessary information can make the model very complex which can increase the model learning time and without any increase in the accuracy level of the model. While on the other hand, removing useful features can make the mode useless because the model might not reflect the exact state of the given problem. The purpose of feature selection is to select only those features from the model which clearly express the state of the model and make it simpler.

1.4.2.1. Forward Feature Selection

A forward feature selection technique reduces the number of feature combinations that are evaluated to figure out the best model with least number of important features. In forward feature selection method, we start with an empty model and on each iteration we select a feature from the number of features available in the model. We evaluate the

current model with all the features in the model and evaluate the feature with gives the best result and then finally apply this feature to this model. We will continue doing this till no feature gives the best model that the currently obtained model. In all, the overall number of model that we evaluate will be quadratic in number.

Parallelizing the forward feature selection is very straight forward. In forward feature selection, each iteration we determine the best feature that needs to be added to the current model. We can parallelize this step by dividing all the features between the numbers of threads and determine the result for each feature in parallel based on the current model and then compare the result of all features with current model and add the best feature to the model.

1.4.2.2. Single Feature Optimization (SFO)

Single feature optimization works by storing the coefficients of the current model and then selects any feature from the number of features available. For the selected feature, SFO optimizes the values of its coefficient and keeping the coefficients of other feature constant, it reruns the complete model. Now this newly created model will have a selected feature in it. We then, begin with this model for next feature selection and optimization. As we are only running the model ones for each feature, SMO will build only liner number of models in comparison to the quadratic number of models builds by forward feature selection method. The drawback of this approach is that the model obtained is approximate.

1.4.3. Hardware Accelerators

CPUs are general purpose processors which are designed to perform a wide variety of calculations easily. CPUs circuit and low level code are designed to be adaptable to call kinds of calculation because of which they are not that efficient. Special hardware is available to perform specific types of computation like GPU is a specialized chip that can perform graphics operations several times faster than a CPU.

In support vector machine, complexity per iteration for calculating kernel operation is in order of number of data instances times features. For a large number of features, complexity of kernel operations can be very high and time consuming. If we can use hardware accelerator which can perform hundreds of multiply accumulate operations at a given time, kernel computation will be very fast.

We can attach the hardware accelerator to the computer and when we run the computationally intensive algorithm like the Support Vector Machine, the computer will send the kernel evaluation part to the hardware accelerator for faster processing and hardware accelerator will return back the computed result to the computer.

CHAPTER 2

PARALLEL SUPPORT VECTOR MACHINES USING SMO[1]

[1] Naman Fatehpuria, Mustafa Veysi Nural, John A. Miller, to be submitted to IEEE Big Data Services 2015.

ABSTRACT

We present an algorithm for Support Vector Machines that can be parallelized effectively. The Algorithm scales up nicely on very large datasets of million training points. Instead of analyzing and optimizing the whole training set in to one support vector machine, the data is split into subsets and each subset is optimized independently on different Support Vector Machine. The result from each Support Vector Machine are then combined to get the trained Support Vector Machine. The high performance is due to low overhead communication between the different Support Vector Machines. In this paper, the runtime performance of the algorithm is tested on a dataset of more than 8 million instances with a speed up of about 20 fold.

2.1. Introduction

Recently, there has been a surge of interest in Support Vector Machines (SVMs) [1]-[3]. SVMs have empirically been shown to give good performance on a wide variety of problems such as handwritten character recognition, face detection, pedestrian detection, and text categorization. Training of a support vector machine for a large dataset for the purpose of classification is computationally very intensive. Many methods have been proposed to reduce the training time, such as Vapnik chunking approach [1], smaller quadratic sub problem method by Osuna [5], Joachims'SVM light [6] which introduces shrinking and caching of kernel techniques, Platt's Sequential Minimal Optimization (SMO) algorithm [7], Modification of Sequential Minimal Optimization [8], Pegasos [28], TRON, SVMperf, and primal coordinate descent implementation [25]. Despite the extensive research that has been done to accelerate SVM training, it is still very significant for large training sets.

In this paper we discuss a parallel approach for implementing support vector machine. The parallel algorithm presented is based on the Modified Sequential Minimal Optimization (SMO) technique [8]. The performance gain of our algorithm is largely due to the way parallelism is handled on a large dataset. It completely eliminates the dependency on common memory data structures for individual threads, so each thread executes independently without blocking the execution of other threads. This approach drastically reduces the training time for large datasets which is linearly proportional to the number of cores on the running machine. In addition, our parallel support vector machine implementation has an increased accuracy of 8% compared to the sequential implementation. We evaluated the performance of our parallel support vector machine

algorithm with extensive experiments on a dataset containing up to 8 million instances and 150 features. For our experiments, we have compared the runtime of our algorithm with the sequential Modified SMO reduction in training time. We discuss the major factors that influence the runtime of the algorithm.

Additionally, we discuss a framework which provides necessary means to integrate our SVM implementation in a large scale analytics pipeline.

Section 2.2 presents background information on support vector machine and Sequential Minimal Optimization. It also explain the terminology and equations that will be used in the rest of the paper. Section 2.3 discusses the implementation details of sequential implementation technique that is based on the Modified SMO and parallel implementation technique to speed up support vector machine. Framework is explained in sector 2.4. Section 2.5, we discuss the related work. Section 2.6, we present the algorithm for both sequential and parallel implementation. We present experiment results and performance evaluation in section 2.7, and we conclude with a summary of results presented and a discussion of future research opportunities in section 2.8.

2.2. Background

2.2.1. Support vector machine

In this section, we will briefly explain standard support vector machines. For linear support vector machines, suppose we have a training data of size *N*.

$$\{ (\boldsymbol{x_1}, y_1), (\boldsymbol{x_2}, y_2), \ldots\ldots, (\boldsymbol{x_n}, y_n) \}$$

Where $\boldsymbol{x_i} \in R^d$ and $y_i \in \{\pm 1\}$. We need to find a linear separating hyperplane which has the maximum separating margin. The Support Vector Machine training problem can be formulated as follows:

$$min_{w,b} \ \frac{1}{2} \ \boldsymbol{w} \cdot \boldsymbol{w}$$

$$\text{Subjected to} \quad y_i \ (\boldsymbol{w} \cdot \boldsymbol{x_i} \ - \ b) \ \geq \ 1$$

where $\boldsymbol{w}$ is perpendicular to the hyperplane. For the above constrained optimization problem we use Lagrange multipliers. From the Karush-Kuhn-Tucker (KKT) conditions, a Lagrange multiplier α_i will be introduced for every data point. The resulting problem for support vector machines is the following:

$$min_{\alpha} \frac{1}{2} \sum_{i=1}^{n} \alpha_i \, y_i \, \boldsymbol{x_i} \ \sum_{i=1}^{n} \alpha_i \, y_i \, \boldsymbol{x_i} \ - \ \sum_{i=1}^{n} \alpha_i$$

$$\text{Subject to } \sum_{i=1}^{n} \alpha_i \, y_i = 0 \ \text{ and } \ \alpha_i \ \geq \ 0$$

and the equation of hyperplane is given by

$$\boldsymbol{w} \cdot \boldsymbol{x} - b = 0$$

2.2.2. Sequential minimal optimization (smo)

Sequential Minimal Optimization was first proposed by Platt [7], which introduced the concept of limiting the working set to size 2 only so the equality constraint can be used to eliminate one of the two Lagrange multipliers. Therefore, the optimization problem at each step can be reduced to a quadratic minimization in only one variable.

The optimality condition can be tracked through the following vector which is constructed as the algorithm progresses.

$$F_i \ = \ \sum_{i \, = \, 1}^{n} \alpha_i \, y_i \, \boldsymbol{x_i} - \ y_i$$

We can partition the training data points into five sets, which can be represented as following:

$$I_0 \equiv \{ i : 0 < \alpha_i < C \}$$

$$I_1 \equiv \{ i : y_i = +1, \alpha_i = 0 \}$$

$$I_2 \equiv \{ i : y_i = -1, \alpha_i = C \}$$

$$I_3 \equiv \{ i : y_i = +1, \alpha_i = C \}$$

$$I_4 \equiv \{ i : y_i = -1, \alpha_i = 0 \}$$

Based on the data point set and optimality condition, the following condition implied.

$$i \in I_0 \cup I_1 \cup I_2 \implies F_i \geq b$$

$$i \in I_0 \cup I_3 \cup I_4 \implies F_i \leq b$$

To check if condition holds, we define

$$b_{up} = \min \{ F_i : i \in I_0 \cup I_1 \cup I_2 \}$$

$$b_{low} = \max \{ F_i : i \in I_0 \cup I_3 \cup I_4 \}$$

$$I_{up} = \arg min_{i \in I_0 \cup I_1 \cup I_2} F_i$$

$$I_{low} = \arg max_{i \in I_0 \cup I_3 \cup I_4} F_i$$

The KKT condition implies $b_{up} \geq b_{low}$. Given the first α_i , these comparisons will automatically find the second α_i for joint optimization in SMO [7].

2.3. Implementation techniques

2.3.1. Sequential implementation

Sequential implementation of support vector machine is based on the Modified Sequential Optimal Minimization approach introduced by Keerti [8]. The idea of the

modified SMO is to optimize two α_i associated with the b_{up} and b_{low} at each step as shown in the below equations.

$$\alpha_1^{new} = \alpha_1^{old} + y_{I_{low}} \frac{(b_{up} - b_{low})}{\eta}$$

$$\alpha_2^{new} = \alpha_2^{old} + s\,(\alpha_1^{old} - \alpha_1^{new})$$

$$\eta = 2\,\kappa(\boldsymbol{x_1}, \boldsymbol{x_2}) - \kappa(\boldsymbol{x_1}, \boldsymbol{x_1}) - \kappa(\boldsymbol{x_2}, \boldsymbol{x_2})$$

$$s = y_1\, y_2$$

α_1^{new} And α_2^{new} must be clipped to a valid range

$$0 \leq a_i \leq C$$

After updating the values of α_i, we need to update the optimality condition vector F_i for all data points. This can be done as follows.

$$F_i^{new} = F_i^{old} + (\alpha_1^{new} - \alpha_1^{old})\, y_1\, \kappa(\boldsymbol{x_1}, \boldsymbol{x_i}) +$$

$$(\alpha_2^{new} - \alpha_2^{old})\, y_2\, \kappa(\boldsymbol{x_2}, \boldsymbol{x_i})$$

Based on the updated values of F_i ,b_{up}, b_{low} and the associated index I_{up} and I_{low} are updated again. The updated values are then used to choose another two new α_i to optimize the next step.

2.3.2. Parallel implementation

We developed the parallel implementation of Support Vector Machine, based on the modified SMO algorithm. SMO spends most of its computation time in updating F_i and this update is performed for every instance of a dataset.

In our parallel approach, we are dividing the entire training data into equally partitioned subsets based on the number of cores in the system. Each smaller training

subset is then handed over to a separate Support Vector Machine. If we are dividing our training data into n subsets, then we will initialize n Support Vector Machines and provide each machine with its own training data subset. By using this approach, n subsets will be trained in parallel by its individual Support Vector Machines.

This approach will determine a subset of F_i for each Support Vector Machines independently in parallel, therefore significantly reducing the computational cost of calculating F_i, which in turn reduces the training time of the complete model.

Each Support Vector Machine will not only calculate F_i for the subset but will also calculate b_{up}, b_{low} and $\boldsymbol{w}$. As a result, each Support Vector Machines will provide a complete model for the provided subset of training data.

The value of b_{up}, b_{low} and $\boldsymbol{w}$ obtained from each support vector machine are not the final values of the model for the whole dataset but are the values from sub models that we have trained in parallel each covering a subset of data. To compute the values of the complete model, we take the mean of all the b_{up},b_{low} and $\boldsymbol{w}$ of all the sub models and this averaged value will give the final model that covers the whole dataset.

2.4. Big data analytics framework

Our SVM is implemented as a part of the ScalaTion [27] framework which supports multi-paradigm modeling.

ScalaTion provides many analytics techniques for optimization, clustering, predicting, etc., which could be easily integrated in a large scale data analysis pipeline. Additionally, ScalaTion supports discrete and continuous event simulation.

ScalaTion is organized in three major packages. The analytics package includes implementations of major analytics algorithms which could be categorized under four types: predictors, classifiers, clusterers and reducers. Additionally, the graphalytics package provides implementations for graph based analytics. Optimization packages (minima and maxima) provide algorithms for optimization and implement major optimization paradigms such as Integer Programming and Simplex method.

Finally, the simulation packages provide simulation engines for a variety of different modeling paradigms. Currently, ScalaTion has implementations for tableu, event, process, activity and state oriented models in addition to system dynamics. ScalaTion also has 2D visualization support.

ScalaTion, coded in the JVM based Scala language, makes use of Scala's native parallelism support via `.par` methods in addition to distributed architectures such as Akka. In Scala, the following code snippet performs a sequential sum operation of every element in a list with a constant number.

```
list.map(_ + 42)
```

To run the same operation in parallel one simply runs the following code to execute the same operation in parallel.

```
list.par.map(_ + 42)
```

ScalaTion extends the same paradigm and provides transparent parallel implementations of analytics algorithms such as SVM discussed in this paper. Using ScalaTion, one could easily switch between sequential and parallel implementations of an algorithm by simply adding .par to the import statement of the class or package.

There are also other modeling environments such as Weka [26] sharing similar functionality with ScalaTion. Weka is a popular data analytics and machine learning platform written in Java. Weka provides a very intuitive user interface that allows pre-processing of data in addition to visualization. Weka also provides a Java API for programmatic access to the algorithms. In contrast to ScalaTion, Weka does not include tools for optimization or simulation. By providing an integrated approach, ScalaTion reduces the cost of development time for a multi-paradigm modeling task.

2.5. Presentation of algorithm

2.5.1. Smo algorithm

1. Initialize $\alpha_i = 0$, $F_i = y_i$, $b_{up} = -1.0$, $b_{low} = 1.0$, I_{up} = Any Index of Class +1, I_{low}= Any Index of class -1, fill $I_1 \; and \; I_4$ with training data indexes for $i = 0 \; to \; n$
2. Select two indexes i_1 and i_2 from the training data to do the joint optimization.
3. Train these two indexes i_1 and i_2 and calculate new value of $\alpha_1 \; and \; \alpha_2$.
4. Update w weight vector based on the value of two indexes i_1 and i_2 and difference between the new and old value of $\alpha_1 \; and \; \alpha_2$.
5. Update F_i for i in I_0 using new Lagrange multiplier $\alpha_1 \; and \; \alpha_2$. Also, update F values for i_1 and i_2.
6. Update I_0, I_1, I_2, I_3, I_4 for both i_1 and i_2.
7. Repeat steps 2 to 6 until $b_{up} \geq b_{low}$.

In the SMO implementation, we will start by initializing the α_i, F_i, $\boldsymbol{w_i}$, b_{up}, b_{low}, I_{up}, I_{low}. This algorithm solves the optimization problem for the weight vector w and the threshold b for the model $\boldsymbol{w} \cdot \boldsymbol{x} - b$. Initially, we will fill the set I_1 and I_4 as $\alpha_i = 0$.

We will start by iterating through all the training data points, using each data point i_1 we will update b_{up}, b_{low}, I_{up}, I_{low}. Then we will check for optimality using the current b_{up}, b_{low} and if violates the condition $b_{up} < b_{low}$, we will find another data point i_2 to do the joint optimization with i_1.

In step 2, we will optimize by replacing old values of Lagrange multipliers with new values of Lagrange multipliers. While doing the optimization, we will update the $\boldsymbol{w}$ weight vector to reflect changes in the Lagrange multipliers and update the error cache F_i for i in I_0 using new Lagrange multipliers. We will also update F values for i_1 and i_2.We will repeat this process until $b_{up} < b_{low}$.

2.5.2. Parallel smo algorithm

The Parallel SMO is based on the sequential SMO algorithm.

1. Divide the training data into subsets say p based on the number of cores in the system, to get the optimal speedup.
2. Initialize p support vector machines and pass each of the p support vector machine a sub set of the training data.
3. Run each support vector machine in parallel and train each of them independently.

4. Initialize $a_i = 0$, $F_i = y_i$, $b_{up} = -1.0$, $b_{low} = 1.0$ $I_{up} = Any\ Index\ of\ Class + 1$, $I_{low} = Any\ Index\ of\ Class - 1$, fill $I_1\ and\ I_4$ with training data indexes for $i = 0\ to\ n$ for each support vector machine.
5. Select two indexes i_1 and i_2 from the training data to do the joint optimization for each support vector machine.
6. Train these two indexes i_1 and i_2 and calculate new value of $\alpha_1\ and\ \alpha_2$ for each support vector machine.
7. Update w weight vector based on the value of two indexes i_1 and i_2 and difference between the new and old value of $\alpha_1\ and\ \alpha_2$ for each support vector machine.
8. Update F_i for i in I_0 using new Lagrange multiplier $\alpha_1\ and\ \alpha_2$. Also, update F values for i_1 and i_2 for each support vector machine.
9. Update $I_0,\ I_1,\ I_2,\ I_3,\ I_4$ for both i_1 and i_2 for each support vector machine.
10. Repeat set 5 to 9 until $b_{up} \geq b_{low}$ for each support vector machine.
11. Each support vector machine will provide update value of $F_i,\ b_{up}, b_{low}$ and $\boldsymbol{w}$ for each trained sub set of training data.
12. Average value of b_{up}, b_{low} and $\boldsymbol{w}$ form all the subset will provide the complete trained model.

In the parallel implementation of Support Vector Machines, we will start by dividing the training data set into smaller subsets and will initialize p Support Vector Machines and each Support Vector Machine will get a subset of the training dataset. Each Support Vector Machine will run on a different thread, so that each Support Vector Machine will run independently of each other and therefore reduces the overall training time of the

complete model. Each Support Vector Machine will provide a complete trained model for the corresponding subset. We will average the results obtained from each Support Vector Machine to obtain the trained model for the complete dataset.

As each subset of training data is trained in parallel on independent threads, significant time can be saved during the training due to having no overhead between parallel SVM executions.

2.6. Performance evaluation

In this section, we evaluate the runtime performance and accuracy of our sequential and parallel implementations of our SMO algorithms, and compare the results with existing implementations of Support Vector Machine.

We have created a synthetic dataset with various numbers of instances and numbers of features for testing. The datasets are created as follows: We created two spheres with forty percent overlap between the spheres. The First sphere contains positive data instances and the second sphere contains negative data instances. Instances are uniformly distributed throughout both spheres. In this way, we get a nice overlap of both types of data points. Additionally, we have compared our results with the standard datasets that are used previously to test the Support Vector Machines. These data sets include a9a, real_sim and rcv1 are from [29]. All experiments were run on a Linux server with 12 core, 24 hardware threads Intel Xeon E5-2620 dual CPU running @ 2GHz and 128G of DDR3 RAM. The sequential and parallel algorithms were written in Scala version 2.10 as part of the ScalaTion framework.

To measure the running time and accuracy of our sequential and parallel algorithms, we have used 5-fold cross validation technique [24] on all the datasets against we which we have compared our results. In 5-fold cross validation, dataset is randomly partitioned into five equal size datasets. Out of five subsets, a single subset is retained as validation dataset for testing the model and the remaining four subsets are used as training datasets. The cross validation process is then repeated five times, with each of the five subsets used exactly once as validation data. The five results from the each run is then averaged to get the final result. By following this approach, we make sure that all instances are used for both training and validation.

The results comparing the accuracy of our sequential and parallel implementations and the Cascade SVM approach [12] are presented in Table 2.1. We have used our own implementation of cascade approach described in [12] and tested the results with our real datasets with 5-fold cross validation. Results clearly indicate that the accuracy of our parallel SVM implementation is significantly higher than both the sequential and cascade implementations.

Table 2.1: The effect of data sets on the accuracy of classification. Accuracy is measured in percentage.

Dataset Name	**Sequential**	**Parallel**
a9a	76.3	76.3
real_sim	98.0	73.7
rcv1	47.5	52.1

Experiments are performed to measure the speed up achieved on the parallel implementation compared to the sequential implementation. The Speed up ratio is shown in Figure I. The ratio will increase with respect to the number of data instances resulting in a linear speed up. As shown in the Figure I, we can get up to 20 fold speed up with a data set of size 8 million instances and 50 features.

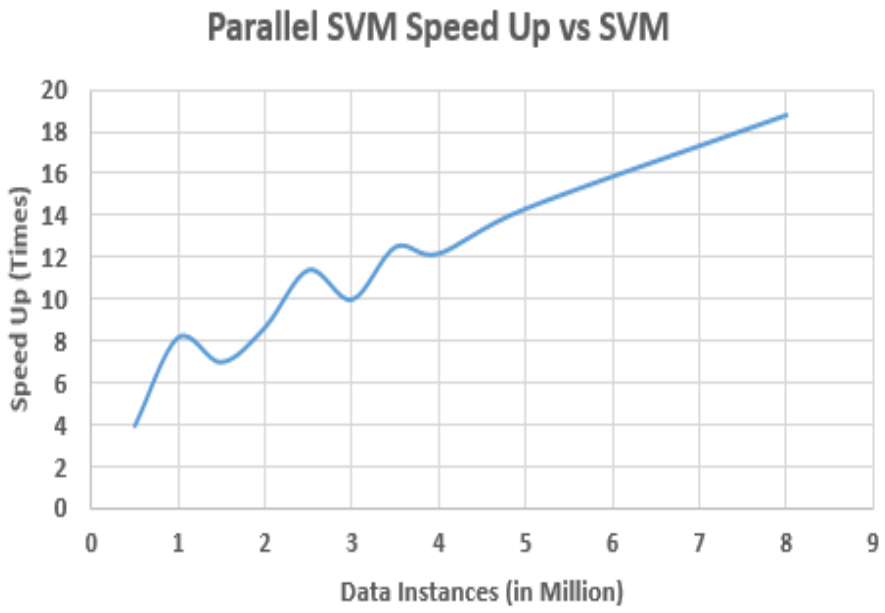

Figure 2.1: Speed up achieved on Parallel SVM in comparison to SVM.

Many experiments are performed to compare the running time on sequential, parallel and cascade algorithms on different number of data instances with different number of features. As shown in Figure 2.1, in which number of features are kept constant to fifty. We can see that sequential implementation takes more than 800,000 millisecond but on the other hand parallel SVM takes only about 40,000 millisecond, which about twenty

folds faster as mentioned above. Cascade approach is much better than our sequential SVM but parallel SVM is out performs cascade approach.

Similar experiment to measure the runtime differences are performed on same synthetic data set but with more number of features to make sure speedup is still maintained as with the increase in the number of features. Results for the runtime with 150 features are shown in Figure 2.3, which clearly indicates that runtime is much better for parallel SVM than sequential and cascade approach.

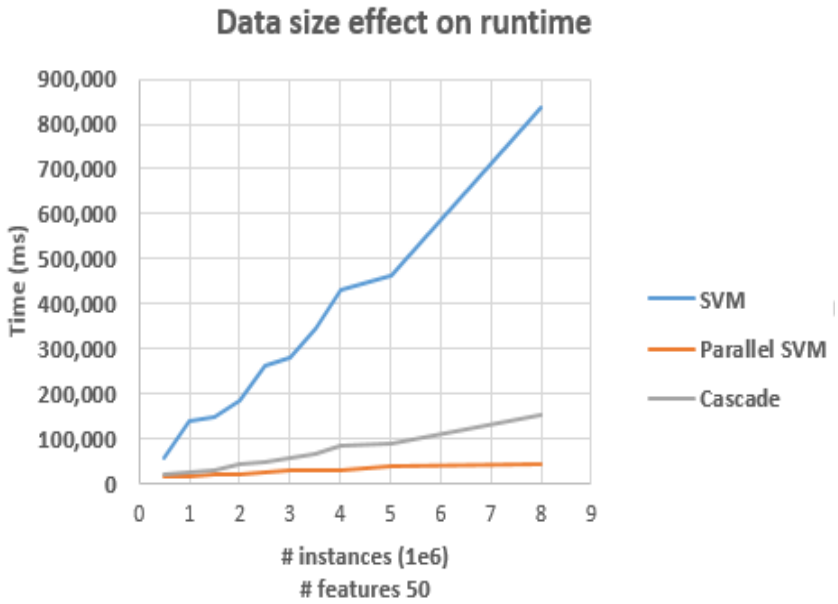

Figure 2.2: Runtime for 50 Features. The effect of number of data instances on the runtime.

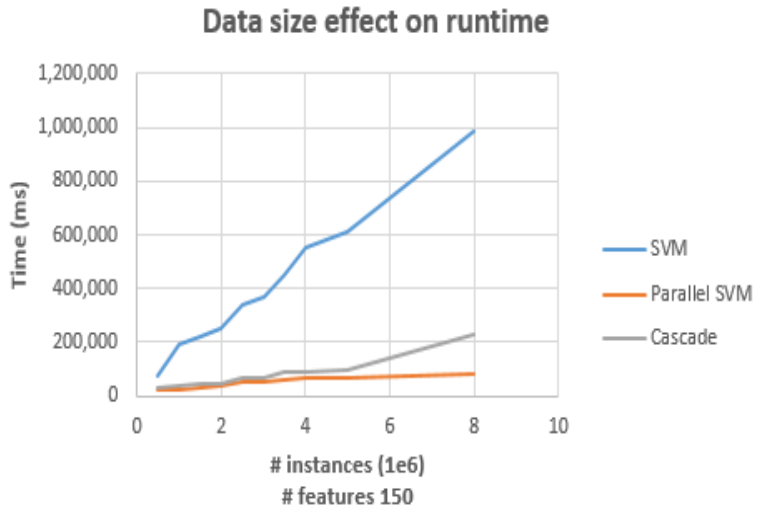

Figure 2.3: Runtime for 150 Features. The effect of number of data instances on the runtime.

Results for parallel SVM are not only compared to the sequential SVM and the cascade approach, but we also performed the experiments on the standard data sets that are used to verify the accuracy of support vector machine. We have compared the results of parallel SVM with the already implemented techniques and results that are explained in [25]. Table 2.2 provides us with the data statistics for the data sets against which we have compare our results.

Table 2.2: Data statistics. Data sets that are used to compare the results.

Data set	Instances	Features	# Non Zeros
a9a	32,561	123	451,592
real-sim	72,309	20,958	3,709,083
rcv1	677,399	47,236	49,556,258

2.7. Related work

The problem of making support vector machines parallel have been studied for decades and yet, it is still an active research area. In [12], parallel approach for support vector machine based on SMO is presented. Author, in this paper has suggested a technique in which instead of analyzing the whole training set in one optimization set, the data are split into subsets and optimized separately with multiple SVMs. The partial results are combined and filtered again in a cascade of SVMs, until the global optimum is reached. In this, set of support vectors from two SVMs are combined and then optimized again on a new SVM, this combining and optimization is continued until only one set of vectors is left but to reach global optimum the result of the last layer is fed back into the first layer. Even though, this approach is fast as it divides the data and get rid of most of non-support vectors but it still spends lot of time combining the results from different SVMs and re-optimization them many number of times till the global optimum is reached and this slows down the total training time of the support vector machine.

2.8. Conclusions and future work

In this paper, we have presented a novel algorithm for parallel support vector machine that is superior to or at least competitive to the state of the art algorithm on very large data set adhering to big data research. In our parallel approach, we have equally partitioned our large data in to smaller subsets and we have trained each subsets independently. As training of subsets are not dependents on the result of other subsets, which resulted in the significant reduction in the training time of parallel implementation. We have also found the increase in the overall accuracy of the results

in parallel SVM. We have demonstrated our results with extensive experiments on both synthetic and real standard data sets available for support vector machine. Scala code support vector machine implementation can be found in analytics package of ScalaTion. For more details, please refer to the supplement.

Future work includes extending the parallel support vector machine to distributed support vector machine which will help in gaining further speed up in the training time of large data instances with many number of features. Adapting the algorithm to work on data sets stored on disk or in a distributed environment would allow to handle ever larger training data sets.

CHAPTER 3

SUMMARY

In summary, we have presented a novel algorithm for parallel support vector machine that is superior to or at least competitive to the state of the art algorithm on very large data set adhering to big data research. In our parallel approach, we have equally partitioned our large data in to smaller subsets and we have trained each subsets independently. As training of subsets are not dependents on the result of other subsets, which resulted in the significant reduction in the training time of parallel implementation. We have also found the increase in the overall accuracy of the results in parallel SVM. We have demonstrated our results with extensive experiments on both synthetic and real standard data sets available for support vector machine. Scala code support vector machine implementation can be found in analytics package of ScalaTion. For more details, please refer to the supplement.

Future work includes extending the parallel support vector machine to distributed support vector machine which will help in gaining further speed up in the training time of large data instances with many number of features. Adapting the algorithm to work on data sets stored on disk or in a distributed environment would allow to handle ever larger training data sets.

REFERENCES

[1] Vapnik, V. N., “Estimation of dependences based on empirical data” (Vol. 40). New York: Springer-Verlag. (1982).

[2] Vapnik, V. N., “The Nature of Statistical Learning Theory”, Springer-Verlag, (1995).

[3] Burges, C. J. C., "A Tutorial on Support Vector Machines for Pattern Recognition," submitted to Data Mining and Knowledge Discovery (1998).

[4] Bryan Catanzaro, Fast Support Vector Machine Training and Classification on Graphics Processors

[5] Osuna, E., Freund, R., Girosi, F., "Improved Training Algorithm for Support Vector Machines," Proc. IEEE NNSP ’97, (1997)

[6] SVMlight Support Vector Machine, Author: Thorsten Joachims

[7] Platt, J., "Sequential minimal optimization: A fast algorithm for training support vector machines". In Technical Report MST-TR-98-14. Microsoft Research, (1998)

[8] Keerthi, S. S. et al., "Improvements to Platt's SMO algorithm for SVM classifier design." Neural Computation 13.3 (2001): 637-649.

[9] Cao, L. J. et al, "Parallel sequential minimal optimization for the training of support vector machines." Neural Networks, IEEE Transactions on 17.4 (2006): 1039-1049.

[10] Peng, P., Ma, Q. L., & Hong, L. M. ,“The Research of the Parallel SMOalgorithm for solving SVM”, In Machine Learning and Cybernetics, 2009 International Conference on (Vol. 3, pp. 1271-1274).

[11] Catanzaro, B., Sundaram, N., & Keutzer, K., “Fast support vector machine training and classification on graphics processors”, In Proceedings of the 25th international conference on Machine learning (pp. 104-111). ACM.

[12] Graf, H. P., Cosatto, E., Bottou, L., Dourdanovic, I., & Vapnik, V., “Parallel support vector machines: The cascade svm”, In Advances in neural information processing systems (pp. 521-528). (2004)

[13] Guest editorial, “vapnik-chervonenkis (vc) learning theory and its applications”, Neural Networks, IEEE Transactions on, vol.10, no.5, pp.985, 987, Sept. 1999

[14] Chen, P. H., Fan, R. E., & Lin, C. J, “A study on SMO-type decomposition methods for support vector machines”, Neural Networks, IEEE Transactions on, 17(4), 893-908. (2006).

[15] Cao, L. J. et al, "Parallel sequential minimal optimization for the training of support vector machines." Neural Networks, IEEE Transactions on 17.4 (2006): 1039-1049.

[16] Dong, J. X., Krzyżak, A., & Suen, C. Y.,“A fast svm training algorithm”, In Pattern Recognition with Support Vector Machines (pp. 53-67). Springer Berlin Heidelberg. (2002).

[17] Pacheco, P.S., Parallel Programming with MPI, San Francisco, Calif.: Morgan Kaufmann Publishers, 1997

[18] Guang, B.H., et al., “Fast Modular Network Implementation for Support Vector Machines”, IEEE Transactions on Neural Networks, Vol. 16, No. 6, Nov. 2005, 1651-1663

[19] Zanghirati, G., Zanni, L., “A parallel solver for large quadratic programs in training support vector machines,” Parallel Computing, Vol. 29, No. 4, pp. 535-551, 2003

[20] Dong, J. X., Krzyżak, A., & Suen, C., “A fast parallel optimization for training support vector machine”. In Machine Learning and Data Mining in Pattern Recognition, (pp. 96-105), Springer Berlin Heidelberg. (2003)

[21] Collobert, R., Bengio, S. and Bengio, Y., “A parallel mixture of SVMs for very large scale problems,” Neural Computation, Vol. 14, No. 5, pp. 1105 – 1114, 2002

[22] Chang, C. C., & Lin, C. J., “LIBSVM: a library for support vector machines”. ACM Transactions on Intelligent Systems and Technology (TIST), 2(3), 27. (2011)

[23] Shevade, S. K., Keerthi, S. S., Bhattacharyya, C., & Murthy, K. R. K., “Improvements to the SMO algorithm for SVM regression”. Neural Networks, IEEE Transactions on, 11(5), 1188-1193. (2000).

[24] Cross-validation, http://en.wikipedia.org/wiki/Cross-validation_(statistics)

[25] Hsieh, C. J., Chang, K. W., Lin, C. J., Keerthi, S. S., & Sundararajan, S., “A dual coordinate descent method for large-scale linear SVM”. In Proceedings of the 25th international conference on Machine learning (pp. 408-415). ACM. (2008)

[26] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H., "The WEKA data mining software: an update", ACM SIGKDD explorations newsletter, 11(1), 10-18. (2009)

[27] Miller, J. A., Han, J., & Hybinette, M., "Using domain specific language for modeling and simulation: Scalation as a case study", In Proceedings of the 2010 Winter Simulation Conference (WSC), (pp. 741-752). IEEE. (2010).

[28] Shalev-Shwartz, Shai, et al. "Pegasos: Primal estimated sub-gradient solver for svm." Mathematical programming 127.1 (2011): 3-30

[29] LIBSVM Data: Classification, http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html