Parallel Support Vector Machine on Apache Spark

Mengyi Zhu

December 9, 2014

1 Abstract

In this project, I will build a convenient parallel support vector machine on Apache Spark. I intend to design a parallel SVM that is easy to use for general purpose. Moreover, since SVM runs on the same data set iteratively, I can take advantage of the RDDs of Spark, which basically can be considered as a shared memory pool. Finally, I will compare my version of SVM to some other SVM algorithms.

2 Introduction

As Big Data becoming more and more popular, machine learning algorithms start to play an important role in computer science. One of the most popular algorithms is the Support Vector Machine, which can be used in data classification and regression. As we all know, SVM is a well-motivated linear learning algorithm with good accuracy. However, since the original SVM algorithm was developed decades ago, it has some limitation on the performance[6]. Generally speaking, because the SVM algorithm reduces the problem of classification to an optimization problem with large size information matrix, SVM's performance suffers severely as the size of the problem increases.

In order to show the performance bottleneck, let's first look at the original SVM algorithm in more detail. Given the problem that we want to learn the mapping $x \to y$ where x is an object with a set of features and Y is the class label. For Simplicity, let's take $x \in R^n$ and $y \in \{\pm 1\}$ first. Besides, let's assume we have training set $\{(x_1, y_1), ..., (x_m, y_m)\}$. Then, our general goal is to find y for any given x. In other words, we want to find a classify function $y = f(x, \alpha)$ where α is the function parameter. Since we only consider the binary classification of y, this is equivalent to finding a hyperplane in space R^n that maximizes the margin between the two classes of data X (y equals 1 or -1) with smallest training error. As a result, Vapnik & Chervonenkis [14] showed that the problem can be mathematically formulated as a quadratic optimization

problem:

$$minP(w,b) = \frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi_i \tag{1}$$

with subject to:

$$1 - y_i(w * x_i + b) \geq 1 - \xi_i, \xi_i \geq 0 \tag{2}$$

where w and b are the characteristics of the desired hyperplane, and ξ is the non-separable error. Once we have w and b, we can represent our classifier function $y = f(x, \alpha) = wx + b$. However, linear classifier as above may be too simple for some problems. More often data analysts have to map x into an inner product space to have richer features. In this case, we are dealing with $\phi(x)$ instead of x in the above equation. Since the dimension of $\phi(x)$ can be large or even unknown, it is very hard to solve the above equation. Fortunately, by applying Representer's Theorem, we have

$$w = \sum_{i=1}^{m}\alpha_i\phi(x_i) \tag{3}$$

so instead of w, we are optimizing α with

$$min_\alpha D(\alpha) = \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j K(x_i, x_j) - \sum_i y_i a_i \tag{4}$$

where

$$\sum_i \alpha_i = 0 \text{ and } 0 \leq y_i a_i \leq C \tag{5}$$

with classifier function:

$$f(x) = \sum_i \alpha_i K(x_i, x) + b \tag{6}$$

The function $K(\bullet, \bullet)$ is called kernel function. After doing this, the original problem becomes a dual problem with Lagrangian multiplier variable α. Finally, this dual problem is solved by using a QP solving method such as Interior Point Method (IPM). As a conclusion, the SVM algorithm's computation really relies on the size of the kernel matrix, which contains all the results by applying kernel function to the entire datasets. Therefore, improvements can be done in either reducing the kernel matrix, partitioning the kernel matrix or parallelizing the QP solving method.

On the other hand, there are several reasons for me to use Apache Spark in this project. First of all, Spark is a current state-of-art distributed system that is widely used and available to any programmers. It can be configured in many different ways to work with various size of problems. For example, Spark can be installed on a single multi-core machine to work on small datasets; Likewise, it can be used in Amazon Clusters on much larger data. In addition, it can also work well with resource manager such as Yarn or Mesos to manage

the job more efficiently. More important, Spark is built based on the concept of programming with Resilient Distributed Datasets(RDD)[16]. RDD can be considered as an immutable collection of partitioned records. They can either be stored in memory or in disk. Therefore, we can cache the popular RDDs in memory so that we can improve the data I/O in iterative processing. As we all know, the nature of most machine learning algorithms such as SVM is an iterative analysis based on the dataset. Therefore, with the help of Spark's RDD, data I/Os can be significantly reduced comparing to other distributed systems.

3 Related Work

There are many reasons for the SVM to be so popular. First of all, SVM is a very general but powerful tool for solving classification and regression problems. Another excellent property of SVM is that it is easy to understand. The model of SVM contains a set of support vectors to indicate the margin between different classes which people can directly get some idea by just looking at the vectors. Besides, the nature of the SVM problem is really a quadratic programming problem which is well studied by mathematicians. However, SVM suffers from both time and space complexity because of this. Without optimization, quadratic programming problems are $O(n^3)$.

The first approach to overcome SVM's scalability issue is to reduce the feature size. In other words, the size of the kernel matrix can have a major effect in the SVM algorithm. Correspondingly, many outstanding extensions or optimizations were raised in order to reduce the matrix size. Among all, Feature selection and feature transformation are the two major approaches in this area[15]. Roughly speaking, feature selection optimizations choose a subset of the features and feature transformation algorithms transfer the original features into a new set of features. For instance, the Principal Component Analysis[11], Singular Value Decomposition [8]and Correlation Based Feature Selection [10] all take this approach.

As a result, many different versions of SVM have been developed, such as libSVM, lightSVM and etc to optimize the original SVM [3]. Among them all, libSVM can be considered the most popular since it provides a set of different extensions of SVM such as C-SVC, nu-SVC for classification and epsilon-SVR and nu-SVR for regression models. However, this approach quickly again hits its bottleneck as it does not change the nature of the QP problem. According to *Vapnick et. al.*, the time complexity of the original SVM is in between $O(n^3)$ and $O(n^2)$ [7]. and even with optimization, the time complexity of libSVM is still bigger than $O(n^2)$.

In addition to the first approach, computer scientists discovered that SVM has the potential to be parallelized efficiently and scales to large problems. Therefore, many effort was made in parallelizing the SVM algorithm and many different versions of parallel SVM were raised in last decade. *Collobert et al.* [5] propose a parallel SVM that cut the data into chunks and each chunk is

trained separately first. Then the separate classifiers are combined into a final single classifier. In 2004, *Graf et al.* [9] propose the Cascade SVM, which improves the idea of "chunking" the data into small sets and train the dataset chunks in a top-down network topology with multiprocessors. A quite different approach was made by *Syed et al.* [13] and was improved by *Caragea et al.* [1]. In their DSVM proposal, the algorithms find the support vectors locally and process them together in a central processing center. In case of the local minimum issue, the central processing center iteratively sends vectors back to local machine for them to improve. Recently, as many parallelization tools have been developed, many scientists propose parallel SVMs based on different parallelization tools. For example, *Edward et al.* [4]develop a parallel SVM called PSVM. The PSVM takes advantage of both parallelization and vector size reduction. First, it applies an Incomplete Cholesky Decomposition(ICF) on the kernel matrix to reduce the size of the problem. Then it uses Open MPI to parallelize the process of solving QP of SVM. At the same time, another paper was published by *Catanzaro et al.* [2] to implement parallel SVM on GPU. All the work above are very efficient in parallelizing the SVM algorithm. However, most of their works are based on a single machine with limited memory. As the cloud system has become more popular, I believe that having a parallel SVM algorithm built on the cloud system can be useful as well.

4 Approaches

Before I explain the implementation of my Parallel SVM, I would like to first clarify the following:

- Apache Spark's Mllib library already contains a SVM algorithm. However, it only provides linear model. Besides, it is marked as experimental in their documentations. In other words, the usage of this Spark's SVM is still limited. Therefore, I insist that my parallel SVM on Spark is still meaningful.
- There is nothing completely new about my algorithm. In fact, what I am trying to do is to learn the novel ideas from current state-of-art SVM algorithms. Then, I try to implement them with Spark using RDDs.

Thus, instead of bringing up some genius ideas to solve the performance problem of SVM, please consider the goal of my project is to actually build something that works and can be used in general purpose.

4.1 Input

As we all know, datasets can have many different formats such as csv, txt and etc.. Different formats of the same dataset thus can take different amount of spaces and the loading time can vary significantly. Therefore, choosing which format to be taken by my parallel SVM algorithm is very important. In my

design, I choose to take the LibSVM format[3]. There are several reasons to do so: 1) LibSVM format only cares about meaningful features for each entry. Therefore, features set to default values are omitted in the file. By doing so, we can save a lot of space and loading time. 2) There is a large set of benchmark datasets available in this format, so I can have more choices to evaluate the performance of the different SVM algorithms.

4.2 QP solving

As I mentioned above, the core of the SVM algorithm is a QP optimizing problem. As far as I know, there are many QP solving methods available. For example, both IPM and Sequential Minimal Optimization (SMO) algorithms can be parallelized and are widely used in different SVMs. According to PSVM [4] and LibSVM [3], IPM has higher time and space complexity in general while SMO is harder to be parallelized. Finally, I choose to implement the SMO optimization method since I think it may perform better in terms of time and space. On the other hand, although people proposed ICF in optimizing IPM, I think the trade-off overhead of the ICF can still be inefficient.

4.2.1 SMO

In original SMO, the QP problem shown in equation 4 breaks into small subproblems, which avoids a time-consuming numerical QP iteration as an inner loop. The space complexity thus is decreased to somewhere between linear and quadratic in terms of the training set size [12]. According to *john et al.*, SMO decomposes the original QP problem into smallest QP sub-problems. As a result, at every sub-problem, SMO only optimizes two Lagrange Multipliers and updates the SVM. In addition, solving for two Lagrange Multipliers can be done analytically. Therefore, the overall computation is relieved at each iteration.

SMO can be divided into two major part: 1)analytically solve for the two Lagrange multiplier 2) heuristically choose which multiplier to optimize:

- At each step, the SMO reduces the sub-problem to solving the following equation:
$$y_1 a_1 + y_2 a_2 = k \text{ s.t. } a_1 \geq 0 \text{ and } a_2 \leq C \tag{7}$$
- There are heuristics for each Lagrange Multiplier: 1) the first multiplier iterates in an outer loop to fix any outbound samples in the training set. 2) the second multiplier maximizes the size of step w.r.t. each outbound sample error.

The pseudocode of SMO is shown in Algorithm 1.

4.3 Parallelization

According to the algorithm shown in 1, the heavy computation part is during updating f and computing b and I. In fact, all those values are calculated based

Algorithm 1 The original SMO

Require: training data $x_1, ..., x_i$, labels $y_1, ..., y_i$
  Initialize: $a_i = 0, f_i = -y_i, \forall 1, ..., l$
  Compute: $b_{high}, I_{high}, b_{low}, I_{low}$
  Update: $a_{I_{high}}, a_{I_{low}}$
  repeat
    Update: $f_1, ..., f_i$
    Compute: $b_{high}, I_{high}, b_{low}, I_{low}$
    Update: $a_{I_{high}}, a_{I_{low}}$
  until $b_{low} \leq b_{high} + 2\tau$

on a certain part of the training set. Therefore, we can partition the training set into parts and calculate subset of f. Once we have the subset f on each separate process, we can calculate the b and I value locally. Then a merging step is required to find the global value of b and I. In fact, the global b_{low} and the b_{high} is just the maximum and the minimum of all local bs. Similarly, global Is can be found in this way as well.

In Spark, as mentioned above, RDDs can store data in a distributed way. Therefore, it is very natural to partition the dataset into each separate clusters. Besides, the calculated result of f, b and I using RDDs can also be cached easily so that updating and merging them are intuitive too.

4.4 Classification

For simplicity, I will only focus on a binary classification implementation. As a result, the model trained by my Parallel SVM can only classify x with labels ± 1. Intuitively, classification predict each x_i separately and thus can be done in parallel.

5 Evaluation

In this section, I will show the evaluation of my parallel SVM comparing to several other SVM algorithms in terms of 1) classification accuracy, 2) the speed performance on different datasets. In this Experiment, I will exam the result on datasets shown in table 1:

Table 1: Datasets Used in Evaluation

name	class	size	feature
svmguide1	2	3089	4
mushrooms	2	8124	112
rcv1.binary	2	20242	47,236
covtype.binary	2	581012	54
news20.binary	2	19996	1,355,191

All those datasets are standard datasets provided by LibSVM in the LibSVM format. The first two are small datasets that can be worked on by any SVM algorithms and does not need further modification. However, the other three are large datasets such that they may not fit into the memory of a single machine. Some of them may not even fit in a moderate disk.

Furthermore, I will list the machine I am working on in table 2:

Table 2: Devices Used in Evaluation

Name	Operating System	Memory	CPU
Macbook Pro	MAC OSX 10.10.1	8 GB at 1600 MHz*2	2.3 GHz Intel Core i7
Linux	Ubuntu 14.04	8 GB at 1600 MHz*2	3.40 GHz Intel Core i7
Amazon Spark Cluster	Amazon EMR*6	7.5 G each	Intel Xeon Processor

The reason I choose these three instances of machine is because I want to test the SVM algorithms based on general usage. Therefore, I picked a general purpose laptop (Macbook), a desktop(Linux) and a cluster (Spark Cluster). Especially for the Spark Cluster, I picked a moderate cluster with 6 virtual machines in instance of M1.Large. For your information, the M1.Large instances are small virtual machines with moderate network speed speed. Therefore, for most of the cases, it does not take advantage from the CPU power and memory access. In fact, a single M1.Large instance can perform much worse than the laptop and desktop without parallelization.

In order to explain the comparison more clearly, I think it is also necessary to point out that all the other SVM algorithms I am comparing to are either written by the original author or from a well developed open source library. As one can already tell, those SVM algorithms therefore are written in different languages. Although the language platform may affect the final measurements, I believe that using their original code is the only way to keep their novel ideas about their algorithms. Furthermore, when the dataset becomes larger, the effect of the language platform becomes really negligible. After all, to make things clear, I would like to list the algorithms I am comparing to in details as follows:

Table 3: Algorithm and Language Platform

Algorithm Name	tested Language
LibSVM	Python
PSVM	C++
Spark SVM	Scala & Java
My Parallel SVM	Scala & Java

In this experiment, I train the model with the entire set and only use one-third randomly shuffled data for testing. In addition, I only record the exact time of training process so there is no dataset loading or logging time.Finally, the experiment results are shown as follows:

Table 4: Time comparison with Dataset svmguide1 on Laptop

Dataset	Algorithm Name	tested machine	Time	Accuracy
svmguide1	LIBSVM	Macbook Pro	0.4195s	99.32%
svmguide1	PSVM	Macbook Pro	118.3s	100%
svmguide1	Spark SVM	Macbook Pro	0.02892s	84.55%
svmguide1	My Parallel SVM	Macbook Pro	0.4832	99.32%

Table 5: Time comparison with Dataset mushrooms on Desktop

Dataset	Algorithm Name	tested machine	Time	Accuracy
mushrooms	LIBSVM	Linux Desktop	1.269s	99.93%
mushrooms	PSVM	Linux Desktop	86.11s	100%
mushrooms	Spark SVM	Linux Desktop	3.081s	90.46%
mushrooms	My Parallel SVM	Linux Desktop	5.078s	98.83%

Since dataset *mushrooms* and *svmguide1* are relatively small datasets, most of the SVM algorithms finished quickly with good accuracy. However, the speed of PSVM is quite slow. This is because in the original PSVM source code, the authors are trying to be informative step by step. As a result, PSVM does a lot of printing and logging during each process. Moreover, according to their comments, the code is still under developing. On the other hands, the Spark SVM and my Parallel SVM are a bit slower than LibSVM because of the overhead of Spark RDDs' actions and transformations.

Table 6: Time comparison with Dataset covtype.binary on Desktop

Dataset	Algorithm Name	tested machine	Time	Accuracy
covtype.binary	LIBSVM	Linux Desktop	more than 3 days	NA
covtype.binary	PSVM	Linux Desktop	more than 3 days	NA
covtype.binary	Spark SVM	Linux Desktop	16.42s	48.79%
covtype.binary	My Parallel SVM	Amazon Spark Cluster	21080.83s	92.05%

In table 6 and 7, as the datasets increase, apparently LibSVM and PSVM hit their bottleneck as they are not able to finish within 3 days. Therefore, I have to kill the process to protect my hardware device. The Spark SVM seems incredibly fast, nevertheless, it only works with linear models which may perform poorly on accuracy as in table 6. However, my Parallel SVM is able to finish the job within hours with Spark Clusters.

In the last table 8, I try to challenge a very large dataset with more than 1 million features. Unfortunately, due to the limitation of resources, I can not go further to see whether the job can be done eventually with my Parallel SVM.

Table 7: Time comparison with Dataset rcv1.binary on Desktop

Dataset	Algorithm Name	tested machine	Time	Accuracy
rcv1.binary	LIBSVM	Linux Desktop	20947.75s	95.75%
rcv1.binary	PSVM	Linux Desktop	more than 3 days	NA
rcv1.binary	Spark SVM	Linux Desktop	7.554s	95.74%
rcv1.binary	My Parallel SVM	Amazon Spark Cluster	12648.5s	90.32%

Table 8: Time comparison with Dataset rcv1.binary on Desktop

Dataset	Algorithm Name	tested machine	Time	Accuracy
news20	Spark SVM	Amazon Spark Cluster	105.07s	100%
news20	My Parallel SVM	Amazon Spark Cluster	more than 3 days	NA

6 Discussion

Currently, most of the parallel SVM algorithms I mentioned in this paper only support binary classification with particular labels. Furthermore, most of those parallel SVMs require the user to know some knowledge about the parallel tool that these SVMs are using. This can be further improved. In fact, in order to develop a more general and simple parallel SVM, a lot more effort is required. For example, when we build a SVM on distributed system, it may affect the feature selection optimization process. Therefore, there can be trade offs between parallelizing the QP solving process and reducing the matrix size. In addition, how we chunk the data into separate machines also significantly changes the number of communication we have to do. Thus, moving SVM onto a distributed system needs a lot careful thinking and thoughtful design.

Another problem I found in both my SVM design and the others is that SVM algorithms tend not to discuss how to use the memory and disk storage efficiently. However, a common problem I faced during my experiment is that there is not enough memory to continue the computation and the whole process slows down significantly. Therefore, another interesting area of future work could be implementing SVM using disk storage and memory interchangeably on a distributed system.

Finally, although SVM is not a perfect tool in every use cases, it is a very useful tool in machine learning and should never be left out. Moreover, it is very important for the user to understand the basic of the SVM algorithm in order to use it in the right place. Machine learning very often works as a black box to solve the problem in large scale, nevertheless, it works better with understanding.

My code can be find at `https://github.com/andy0727/Parallel-SVM-on-Spark.git`

References

[1] C. Caragea, D. Caragea, and V. Honavar. Learning support vector machines from distributed data sources. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 20, page 1602. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005.

[2] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th international conference on Machine learning*, pages 104–111. ACM, 2008.

[3] C.-C. Chang and C.-J. Lin. Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

[4] E. Y. Chang. Psvm: Parallelizing support vector machines on distributed computers. In *Foundations of Large-Scale Multimedia Information Management and Retrieval*, pages 213–230. Springer, 2011.

[5] R. Collobert, S. Bengio, and Y. Bengio. A parallel mixture of svms for very large scale problems. *Neural computation*, 14(5):1105–1114, 2002.

[6] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[7] S. Fine and K. Scheinberg. Efficient svm training using low-rank kernel representations. *The Journal of Machine Learning Research*, 2:243–264, 2002.

[8] G. H. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14(5):403–420, 1970.

[9] H. P. Graf, E. Cosatto, L. Bottou, I. Dourdanovic, and V. Vapnik. Parallel support vector machines: The cascade svm. In *Advances in neural information processing systems*, pages 521–528, 2004.

[10] M. A. Hall. *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, 1999.

[11] I. Jolliffe. *Principal component analysis*. Wiley Online Library, 2005.

[12] J. Platt et al. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

[13] N. A. Syed, S. Huan, L. Kah, and K. Sung. Incremental learning with support vector machines. 1999.

[14] V. Vapnik. *The nature of statistical learning theory*. springer, 2000.

[15] J. Weston, S. Mukherjee, O. Chapelle, M. Pontil, T. Poggio, and V. Vapnik. Feature selection for svms. In *NIPS*, volume 12, pages 668–674, 2000.

[16] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.