Cascading Vector Machines
CME 323 Project

Lan Nguyen Carlos Riquelme Sven Schmit

June 2, 2015

1 Introduction

Support vector machines are a popular tool for both regression and classification tasks. Using the kernel trick it becomes easy to get terrific results in many different prediction tasks. However, while the linear SVM is easy to parallelize (and is implemented in Spark), kernel SVMs do not scale well in the number of observations.

In this paper we investigate approximate kernel SVM methods that do scale in the number of observations. In particular, we implement a Cascading SVM as proposed by [3], and analyze its performance on the MNIST handwritten digit dataset [4] and the Forest Covertype dataset [1].

This paper is organized as follows: In the next section, we introduce the problem more formally. Then we give a brief overview of relevant literature.

We discuss our implementation of Cascade SVM in PySpark in Section 3. Thereafter, we describe its performance, both in run time complexity as well as communication costs. We also compare the performance on the MNIST dataset in Section 5. Finally, we give some concluding remarks that summarizes our findings.

2 Problem

In this section, we describe the kernel SVM problem formally. We consider the problem of binary classification, although our results trivially carry over to multiple classification and regression problems.

Consider a dataset consisting of N (x_i, y_i) pairs, where $y_i \in \{-1, 1\}$ denotes the label and x_i encodes the features. The linear SVM model is then given by

$$\underset{w,b}{\text{minimize}} \quad \sum_{i=1}^{n} \Big(1 - y_i(w^T x_i + b)\Big)_+ + \frac{\lambda}{2}\|w\|_2^2 \tag{1}$$

proposed by [2].

However, it can be shown that the solution w can be written as $w = \sum_i \alpha_i x_i$. Our prediction for feature vector x then is $\hat{y} = \mathbf{sign}(w^T x) = \mathbf{sign}(\sum_i \alpha_i x_i^T x)$. This gives a linear decision boundary. However, we could replace the inner product $x_i^T x$ with a positive semi-definite function $K_{ij} = \kappa(x_i, x) = \phi(x_i)^T \phi(x)$ that implicitly maps the features to a higher dimensional space. This is known as the kernel trick and allows us to easily model complicated decision functions. One can show that, using this kernel trick, we can equivalently solve the dual problem

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \frac{1}{2}\alpha^T Q \alpha - \sum_{i=1}^{n} \alpha_i \\ \text{subject to} \quad & y^T \alpha = 0, \\ & 0 \le \alpha_i \le C, \quad i = 1, \ldots, n, \end{aligned} \tag{2}$$

where $Q_{ij} = y_i y_j K_{ij}$.

Hence, to solve the above optimization problem, we need access to matrix K that contains inner products of feature mapping $\phi(\cdot)$. However, K is an $N \times N$ matrix, which makes it hard to solve the above optimization problems when N grows. This is in sharp contrast to the linear SVM, where one can even use simple stochastic gradient descent, which scales as $\mathcal{O}(Nd)$, or even indepent from N. We will call this the *scaling problem*.

We should not forget that the kernel trick is really powerful, as it allows us to work with infinite dimensional models (for example, when using the Gaussian kernel). Even though we are implicitly working in the regime where $d = \infty$, we only require $\mathcal{O}(N^2)$ work. However, this is still not good enough for our use case, where N^2 is prohibitively large.

2.1 An interesting property

There is a very interesting property of the Kernel SVM algorithm that guides us to a possible solution. For points x_j that are outside the 'margin', the term α_j equals 0. Hence, these points are irrelevant for prediction. Getting rid of these points outside of the margin as early as possible could therefore help us out with the scaling problem, which is the approach taken by [3]. We will elaborate further on this in the next section, which describes our approach.

2.2 Literature overview

But first, we want to give a quick overview of other possible solutions to the scaling problem. The most straightforward approach when N is too large, is to simply fit a kernel SVM on a subset of the data. Throwing away data is a shame, but at least the problem becomes solvable in reasonable time.

There has been quite a large body of work in approximating the kernel matrix K, with a low rank matrix for example [8]. Alternatively, a promising method is to approximate the feature mapping $\phi(\cdot)$ as proposed by [5]. They show that ϕ can be approximated both using sampling from the Fourier basis, as well as randomly partitioning the input space as long as the kernel is shift-invariant.

There are also methods that approximate the solution in other ways, such as in [7].

3 Cascade SVM

The main idea of Cascade SVM [3] is to split the data into manageable subsets, so that local models can be fitted. Once we have the local models, we would like to combine them in a way that guarantees optimality after a few iterations. This approach is common to other distributed versions of Machine Learning algorithms.

The general structure of the algorithm is given in Figure 3. Let N be the size of the data and M be the number machines used. We first assign approximately N/M data points to each machine, and we fit a kernel SVM locally in each of them. After doing so, we have a set of support vectors at each machine together with their weights. Note that these support vectors are not necessarily support vectors of the whole dataset, but only support vectors with respect to the local subset of data that was assigned to their machine. Our hope is to discard as many points as possible among those not lying on the margin —or, say, at least close to the boundary in the projected high-dimensional space.

In the next step, the sets of support vectors found in pairs of adjacent nodes (processors) are merged together and 'passed' to the next layer. In the consequent layer, each node takes as inputs the union of the support vectors from two parent nodes from the previous layer. At this point, each node fits a kernel SVM locally again. A natural idea is to use the weights found in the previous layer as a 'warm start' to speed up the optimization solver in the current layer.

We repeat this train-then-merge process until we reach a layer with a single node. We solve a last kernel SVM on this final node using the union of the support vector sets from the previous layer as inputs. The result of this final training step is a *complete* model (the support vectors and the weight vector found) for the entire dataset. We refer to the entire process described above as one complete *outer* iteration (or one cascade).

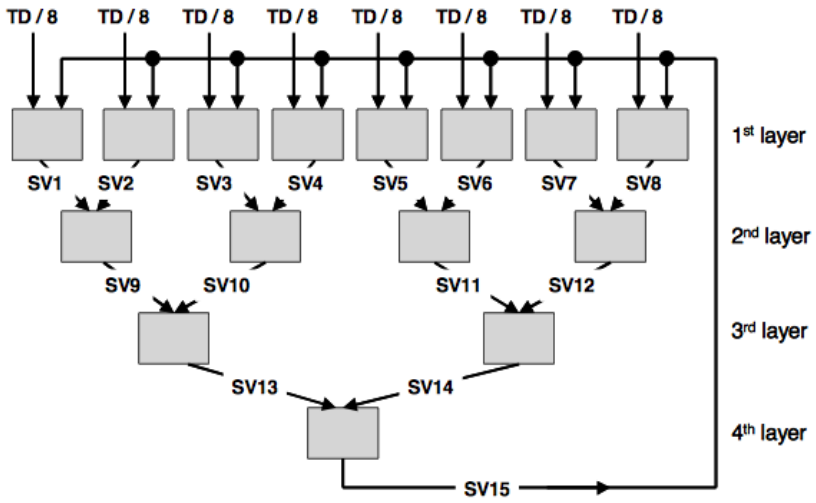

Figure 1: Diagram and text taken from [3]: *"Schematic of a binary Cascade architecture. The data are split into subsets and each one is evaluated individually for support vectors in the first layer. The results are combined two-by-two and entered as training sets for the next layer. The resulting support vectors are tested for global convergence by feeding the result of the last layer into the first layer, together with the non-support vectors. TD: Training data, SVi: Support vectors produced by optimization i."*

The algorithm performs several of these outer iterations. At the end of one iteration, we append the final set of support vectors to each original local partition of the dataset. Then, we start a new iteration with this modified data as inputs. The algorithm stops when the set of support vectors found in each of local partition in the first layer (each leaf node) of a current iteration is included in the final set of support vectors found by the root node in the previous iteration.

Note that the quality of the solutions improves with iterations. Under the assumption that all the support vectors found in one layer are passed to the next layer, Theorem 3. in [3] guarantees that the cascade algorithm converges to the global optimum in a finite number of iterations. In fact, the authors of the paper claim that it usually takes no more than 4 or 5 iterations to converge.

Unfortunately, in practice when dealing with hard problems with wide soft margins, where the number of support vectors is large, it might happen that the union of two sets of support vectors from one layer is too big to fit a local SVM, i.e. the new input size makes solving a local QP infeasible. Our implementation prevents these situations by limiting number of support vectors allowed to be passed between consecutive layers. We fix `nmax` to be the maximum size of input set for each node. This means that each node is only allowed to 'pass' `nmax`/2 support vectors to the next layer. With this constraint, we never encounter a situation where the number of data points in a single node is too large to run a local SVM.

We choose the support vectors to keep uniformly at random. However, other heuristic policies can be implemented to pick which support vectors to retain. One of the options is to use the dual variables of the QP – the weights over the support vectors, to guide the decision about which ones to keep/discard. However, in our experiments this did not improve performance.

4 Implementation details

4.1 Data representation and libraries used

We choose an MLlib object `LabeledPoint` to represent each observation in our data set. This object stores both the feature vector and the label of a sample. We store all data as an `RDD` of `LabeledPoints`, and later process the subset of observations separately on each partition.

Our code is developed in PySpark so that we can use the popular Python machine learning library `scikit-learn`, which depends on `numpy`. The program allows the user to perform binary and multi-class classification, as well as regression tasks. Full code is available on a github repository.[1]

4.2 Complexity

As mentioned in Section 2, the biggest drawback of kernel SVM method stems from the need of computing and storing a kernel matrix, $K \in \mathbb{R}^{N \times N}$,where N is the number of training observations. Apart from this, training a standard kernel SVM is equivalent to solving a quadratic program (QP), which incurs a cost of $\mathcal{O}(dN^{\alpha})$ (where d is the dimension of the feature space, and α is a constant bounded between 2 and 3). This complexity is clearly not scalable with respect to the size of the data.

The core strength of the cascade algorithm lies in the fact that it solves many small-size kernel SVM problems. The method described achieves scalability in the number of sample data, while keeping the linear the complexity with respect to the dimension of the feature space.

Suppose we have M available machines (processors) to our disposal, and solving an QP on a subset of N/M samples is feasible. In the training step, the algorithm runs a number of N/M-sized SVMs in parallel on each of the $L = \log_2(M)$ layers. One pass of a cascade costs $\mathcal{O}(dL\left(\frac{N}{M}\right)^{\alpha}) = \mathcal{O}(d\log(M)\left(\frac{N}{M}\right)^{\alpha})$. Running T iterations of the cascade takes in total:

$$\text{Cost}_{train} = \mathcal{O}\left(dT\log(M)\left(\frac{N}{M}\right)^{\alpha}\right)$$

[1]To access our code go to: `https://github.com/schmit/cvm`

With $\alpha = 3$ and $M = \mathcal{O}(\sqrt(N))$ the complexity is of order $\mathcal{O}(dTN^{3/2}\log(N))$.

Running the entire cascade SVM even on a single core saves some computation costs. In this case we would run all $(2M-1)$ small problems (corresponding to each node in the cascade binary-tree) sequentially, which takes $\mathcal{O}(dT(2M-1)\left(\frac{N}{M}\right)^{\alpha}) = \mathcal{O}(dT\left(\frac{N^{\alpha}}{M^{\alpha-1}}\right))$.

The prediction stage of the SVM takes significantly less time. For a standard SVM this takes $\text{Cost}_{test} = \mathcal{O}(dn_{\text{sv}}N_{test})$, where n_{sv} is the number of support vectors. However, prediction can be performed completely in parallel, and would take $\text{Cost}_{test}/n_{\text{partitions}}$

4.3 Communication

There are three communication types used in our algorithm. At the beginning, we need to shuffle the data to simulate a random split of data across partitions. Repartitioning also prevents any adversary distribution of observations in the processors. This single all-to-all communication takes at most $\mathcal{O}(dN)$ cost.

After this initial step, each iteration of the cascade requires a number of coalesce calls, which merge 2 sets of support vectors found in one layer and passes them to the next layer of the algorithm. If coalesce is done in a smart way, we simply perform $(2M-1)/2$ one-to-one communications. This is because there are $2M-1$ nodes in the cascade binary tree, and half of them send their support vectors to the neighbouring node. Per iteration, this communication type costs $\mathcal{O}\left(\frac{(2M1)}{2}\frac{dN}{2M}\right) = \mathcal{O}(dN)$.

Finally, at the end of the training step, we need to broadcast the model to each processor holding the testing data set. We need a single all-to-one communication to pass all model parameters and the support vectors found to each partition, which requires $\mathcal{O}(d\frac{N}{M})$ cost; with a choice of $M \approx \sqrt{N}$ this is equal to $\mathcal{O}(d\sqrt{N})$.

Overall, the total communication cost required is:

$$\mathcal{O}(dN + TdN + d\frac{N}{M}) = \mathcal{O}(dTN).$$

5 Experiments

We use two datasets to test the performance of our cascade SVM implementation. We consider two metrics to compare with. First, we run kernel SVM on the full dataset, to get an upper bound on the performance. On the other hand, a lower bound on the performance should be given by a simple subsample of the dataset, because we throw away a lot of information.

For timing statistics we consider include the following:

- Loading data
- Training model
- Predicting model on 60k training samples
- Predicting model on 10k test samples

We run all experiments on a single shared memory machine with 512 GB Ram machine with 4 (6) core intel Xeon E7540 2 GHz processors.

5.1 MNIST

First, we consider the MNIST handwritten digit dataset [4], which contains 70,000 images of hand written digits. We split the data randomly into a training set of 60,000 images, and a test set of the other 10,0000 images. This dataset is relatively small, and has been shown that great performance can be achieved. In this case, we use a Gaussian kernel with parameter $\gamma = 0.02$, and we use C-SVC implemented in SKLearn with $C = 1$ in all experiments.

Running Kernel SVM on a single machine on the full dataset leads to a test error of 1.7%, though it takes a long time to train: 4050 seconds. We can improve the run time by sub sampling the data, but this comes at a performance cost. Using only 10,000 samples leads to a test error of 3.5%, trained in 720 seconds. If we down sample even further, to only using 2,000 samples, the test error deteriorates to 6.5%, trained in 124 seconds.

Our implementation of Cascade SVM, fitting models with 2,000 observations achieves a test error of 5.00% after a single pass, which takes 164 seconds. If we fit models with 10,000 observations, we are able to improve to a test error of 2.1% in 964 seconds. While we don't achieve the same accuracy as the full model, we improve dramatically over the sub sampled SVMs without incurring too much of a hit in performance. Using multiple passes through the cascade did not noticeably improve the performance.

However, if we do not restrict the size of the SVMs in later layers of the cascade, we retrieve the optimal test error of 1.7% using 1 pass and 2,000 observations in the first layer. That is, in the first layer we fit SVMs of size 2,000, but if at a later layer there are more than 2,000 observations, we fit a larger SVM instead of throwing away data. This leads to fitting an SVM of size 20,000 to obtain the final result. While this is a big improvement over a single big SVM over all the data, it still costs a lot of extra computation time: note that the final stage can only be run on a single machine.

6 Conclusion

We think that Cascade SVM allows SVM parallelization to an acceptable extent, and leads to fairly good results in some scenarios. The algorithm works best in settings with little noise

and well separated classes, such that relatively few support vectors are needed. However, when the number of support vectors is very large and the capacity of the machines is limited, some of them need to be deleted, which can lead to very bad performance. We haven't figured out an efficient and harmless way to do this, and we rely so far on uniform sampling.

On the other hand, we believe the architecture can be applied to several other kernel methods, like Kernel Logistic Regression, or variations on SVMs such as L1VMs and RVMs [6].

References

[1] C L Blake and C J Merz. Uci repository of machine learning databases, 1998. Robustness of maximum boxes.

[2] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[3] Hans P. Graf, Eric Cosatto, Léon Bottou, Igor Dourdanovic, and Vladimir Vapnik. Parallel support vector machines: The cascade svm. In L.K. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 521–528. MIT Press, 2005.

[4] Yann LeCun and Corinna Cortes. The mnist database of handwritten digits, 1998.

[5] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184, 2007.

[6] Michael E Tipping. Sparse bayesian learning and the relevance vector machine. *The journal of machine learning research*, 1:211–244, 2001.

[7] Ivor W Tsang, James T Kwok, and Pak-Ming Cheung. Core vector machines: Fast svm training on very large data sets. In *Journal of Machine Learning Research*, pages 363–392, 2005.

[8] Christopher Williams and Matthias Seeger. Using the nyström method to speed up kernel machines. In *Proceedings of the 14th Annual Conference on Neural Information Processing Systems*, number EPFL-CONF-161322, pages 682–688, 2001.