Stanford University

CME 323 Final Project

A Distributed Solver for Kernalized SVM

Haoming Li haoming@stanford.edu
Bangzheng He bzhe@stanford.edu

GitHub Repository
`https://github.com/CME323Project/Spark_kernel_svm.git`

June 3, 2015

Contents

1 Introduction

Support vector machine (SVM) is a widely used supervised learning model, originally for binary classification. But it can also be generalized for multiclass classification and regression problems. In this report we only focus on binary classification with SVM, since it's easy to start with and not hard to generalize to other problems. We first look at a sequential algorithm for solving SVM using stochastic gradient descent (SGD) method, and the corresponding parallel algorithm. Then we try to implement the parallel algorithm in Apache Spark. Finally we experiment with several tuning parameters and conclude the project.

1.1 Kernalized Support Vector Machine

Like many machine learning methods, the kernalized support vector machine can be formulated as a convex optimization problem:

$$\min_{w\in\mathbb{R}^d} f(w) = \frac{\lambda}{2}\|w\|_2^2 + \frac{1}{m}\sum_{i=1}^{m} max\{0, 1 - y_i\langle w, \phi(x_i)\rangle\}$$

where $\{(x_i, y_i)|x_i \in \mathbb{R}^d, y_i \in \{-1, 1\}\}_{i=1}^m$. x_i are the training data features and y_i are the corresponding labels taking value either -1 or 1. $\phi(x_i)$ is a mapping function.

This convex optimization objective function $f(w)$ has two parts: a 2-norm regularization that controlling complexity of the model and a empirical loss function measuring the error of the model on the training data. The regularization parameter λ defines the trade-off between minimizing the training error and minimizing model complexity.

For this convex problem, the Lagrange dual objective has the form

$$L_D = -\frac{1}{2}\sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \langle\phi(x_i), \phi(x_j)\rangle + \sum_{i=1}^{m}\alpha_i$$

According to KKT condition, the optimal weights w can be written as a

superposition of training data:

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

Hence we have

$$\langle w, \phi(x) \rangle = \sum_{i=1}^{m} \alpha_i y_i \langle \phi(x_i), \phi(x) \rangle$$

Therefore we need not to specify the mapping $\phi(x_i)$ at all, but require only knowledge of the kernel function:

$$K(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$$

Popular kernels include radial basis function (rbf), d^{th}-Degree polynomial kernels, etc.

$$rbf : K(x_i, x_j) = exp(-\gamma \|x_i - x_j\|^2)$$
$$polynomial : K(x_i, x_j) = (1 + \langle x_i, x_j \rangle)^d$$

1.2 Optimization Methods for SVM

The optimization methods for training SVM models generally fall into three categories. The interior Point Method(IPM) minimizes the dual objective, which is a convex quadratic programming problem and can be solved via the primal-dual interior point method. Sequential Minimal Optimization(SMO) decomposes the quadratic programming problem into an inactive part and an active part. Stochastic Gradient Descent(SGD) method minimizes the primal objective, in which the gradient is approximated by evaluating on a single training sample. We will focus on a SGD algorithm and the corresponding parallelization.

2 Solving SVM using SGD method

In this section we discuss a stochastic gradient descent algorithm proposed in [1]. The single machine version is called S-pack, and the parallel version is called P-pack. We also introduce our own implementation in Spark.

2.1 S-pack SVM algorithm on single machine

The empirical loss in the primal objective:

$$\frac{1}{m}\sum_{i=1}^{m} max\{0, 1 - y_i\langle w, \phi(x_i)\rangle$$

averages the hinge loss among all training examples. According to the idea of SGD, it can be approximated by hinge loss on a single training sample. At iteration t, randomly pick up a sample (x_i, y_i), then we have the sub-gradient:

$$\lambda w - \{ \begin{array}{c} y_i\phi(x_i), y_i \langle w, \phi(x_i)\rangle < 1 \\ 0, otherwise \end{array}$$

For faster learning rate, update the predictor w as below:

$$w \leftarrow (1 - \frac{1}{t})w + \{ \begin{array}{c} \frac{y_i}{\lambda t}\phi(x_i), y_i \langle w, \phi(x_i)\rangle < 1 \\ 0, otherwise \end{array}$$

And to get closer to the optimum, apply a projection after the update:

$$w \leftarrow \min\left\{1, \frac{1/\sqrt{\lambda}}{\|w\|_2}\right\}$$

We can express $w = sv$, where s is a scalar and $v = \sum_i \beta_i\phi(x_i)$. When performing scaling we can just change the value of s instead of modifying the each element of w every time. And we store v as key-value pairs (x_i, β_i).

The pseudo-code of the algorithm is as in algorithm 1.

Training data size m, number of features $d(x \in \mathbb{R}^d)$. Only line 4 takes at most $O(md)$ time, other computations run in constant time. And it was analyzed in [2] that it requires $T = O(1/\lambda\delta\epsilon)$ iterations to get $|f(w) - f(w*)| < \epsilon$ with at least $1-\delta$ probability. Thus runtime is $O(md/\lambda\delta\epsilon)$. Since the optimal $\lambda = O(1/m)$, total runtime can also be written as $O(m^2d/\delta\epsilon)$.

2.2 Parallel algorithm: P-pack

The S-pack algorithm has time complexity in square dependence on the training sample size m. It is not efficient and has limitation on the size of the

Algorithm 1: S-pack SVM

Input: λ, T, training data

Initialize: $H = \emptyset$, $s = 1$, $norm = 0$

for $t = 1, 2, \ldots T$ **do**

  Randomly pick training sample (x, y)

  $y' \leftarrow s \sum_{(x_i, \beta_i) \in H} \beta_i K(x_i, x)$

  $s \leftarrow (1 - 1/t)s$

  if $yy' < 1$ **then**

    $norm \leftarrow norm + 2yy'/\lambda t + (y/\lambda t)^2 K(x, x)$

    if *key* x *is found in* H **then**

      $(x, \beta) \leftarrow (x, \beta + y/\lambda t s)$

    else

      $H \leftarrow H \bigcup (x, y/\lambda t s)$

    if $norm > 1/\lambda$ **then**

      $s \leftarrow s/\sqrt{\lambda norm}$

      $norm \leftarrow 1/\lambda$

return s, H

training data, but the the algorithm itself suggests possible parallelization, named P-pack. We address two key points below.

2.2.1 Distributed Hash Table

As we have seen in the previous section, the bottleneck of the algorithm is the calculation of responses $\langle v, \phi(x)\rangle$, luckily this can be highly parallelized with a distributed storage of the key-value pairs (x_i, β_i).

Now suppose we have p processors and the key-value pairs in H are averagely distributed among all processors. For example, i^th processor holds a subset of H, denoted it by H_i:

$$H_i = \left\{(x_{i_j}, \beta_{i_j})\right\}_{j=1}^{\|H_i\|}$$

then we can calculate

$$\sum_{(x_{i_j}, \beta_{i_j}) \in H_i} \beta_{i_j} K(x_{i_j}, x)$$

locally on each processor at the same time and sum up the results across all processors to speed up the calculation of $\langle v, \phi(x)\rangle$.

Once a key-value pair (x, β) needs to be updated in certain iterations, each processor checks whether the key value x exist in its local hash table H_i. If any processor finds the key, update the corresponding value, otherwise add this new key-value pair (x, β) to the hash table of the least occupied processor.

2.2.2 Packing Strategy

However, with the distributed storage of the training data (x_i, y_i) and key-value pairs (x_i, β_i), in each iteration there are at least one communication request among all processors. This can be optimized by packing r iterations into one round of update, thus reduce the number of communication requests by a factor of $O(r)$.

Denote the training sample picked at iteration t, x_t, and updated weights w_t. Then, the responses $\langle w, \phi(x)\rangle$ from t^{th} iteration to $(t+r-1)^{th}$ can be written in the form

$y'_t = \square \langle v_t, \phi(x_t)\rangle$

$y'_{t+1} = \square \langle v_t, \phi(x_{t+1})\rangle + \square K(x_t, x_{t+1})$

$y'_{t+2} = \square \langle v_t, \phi(x_{t+2})\rangle + \square K(x_t, x_{t+2}) + \square K(x_{t+1}, x_{t+2})$

...

$y'_{t+r-1} = \square \langle v_t, \phi(x_{t+r-1})\rangle + \square K(x_t, x_{t+r}) + \cdots + \square K(x_{t+r-2}, x_{t+r-1})$

Those "blanks" are some complex coefficients. Therefore we can compute the most time consuming part $\langle v_t, \phi(x_t)\rangle, \langle v_t, \phi(x_{t+1})\rangle, \ldots, \langle v_t, \phi(x_{t+r-1})\rangle$ at t^{th} iteration, as well as the pair-wise values $K(x_i, x_j)$ for $t \le i < j \le t+r-1$. Then at the next $r-1$ iteration, there's no communication request across processors to calculate responses y_i for $i = t+1, \ldots, t+r-1$. For more details please refer to [1]

Now instead of picking only 1 random sample, we can pick r training samples at each iteration and calculate the updates as described above, then we only need to implement T/t number of rounds of iterations to achieve the same result, reducing the number of communication requests by a factor of $O(r)$. Although the total bits of communication will not be reduced, even raised a little bit, proper choice of the pack size r will speed up the algorithm significantly due to the reduction of communication frequency.

2.2.3 Implementation of P-pack SVM in Apache Spark

We make a few assumptions about training data.

1. The feature dimension d is not too big, so that a single data point can fit in single machine.

2. Data size m can be large, so that they are supposed to be distributed.

Under these assumptions we represent the training data as RDD[LabeledPoint] in Spark, in which each element has type spark.mllib.regression.LabeledPoint.

The model, which is a set of support vectors and corresponding Lagrangian dual variables, is also distributed so that the computation parallelism is achieved. General RDD is not suitable for this because we want to make small modifications to the model for lots of times. IndexedRDD, developed by Berkeley AMPLab, is currently the best thing we have as a distributed hash table. It can index / modify values given keys more efficiently than general RDD. Our model is represented as an IndexedRDD[(Long, (LabeledPoint, Double)] in which each element is a key-value pair: key is a unique long integer and value is a support vector and corresponding Lagrangian dual variable.

Also notice that in each round of iterations we need to compute pairwise kernel function for r samples. This is parallelized by first generating a list of [(i, j) for i = 1 to r and j = i to r], then distribute this list as an RDD, indicating the allocation of $r(r-1)/2$ computations among workers. In each round of iteration, after broadcasting the sample to all workers each worker can compute their part according to this allocation RDD, thus parallelism is achieved. Algorithm 2 is the pseudo code for P-pack algorithm in Spark framework.

2.3 Analysis of the Algorithm Efficiency

Assume sample data size m, feature dimension d, regularization parameter λ, packing size r, running T iterations (T/r rounds of updates), using p processors.

2.3.1 Computational Cost

In each round:

- Line 5, map and reduce(with combiners) to compute r responses: $O(rd \cdot \frac{m}{p}) + O(rp)$
- Line 6, map to compute pairwise inner-products: $O(\frac{r^2 d}{p})$

Algorithm 2: P-pack SVM in Spark framework

Input: λ, T, r, D (training data rdd)

Initialize:

$H = IndexedRDD(D.map((x, y) => (idx, x, y, alpha = 0))$, $s = 1$, $norm = 0$, $A =$ RDD([(i, j) for i = 1 to r and j = i to r])

for $t = 1, 2, \ldots, T/r$ **do**

  Randomly pick r samples $(idx_1, x_1, y_1), \ldots, (idx_r, x_r, y_r)$, broadcast to all processors

  for $k = 1, \ldots, r$ **do**

    $y'_k \leftarrow H.map(h => h.y * h.\alpha * K(x_k, h.x)).reduce(+)$

  $pair \leftarrow A.map((u, v) => K(x_u, x_v)).collect()$

  $LocalSet \leftarrow \emptyset$

  for $k = 1, \ldots, r$ **do**

    $t' \leftarrow t \cdot r + k$; $s \leftarrow (1 - 1/t')s$

    for $l = k + 1, \ldots, r$ **do**

      $y'_l \leftarrow (1 - 1/t')y'_l$

    if $y_k y'_k < 1$ **then**

      $norm \leftarrow norm + 2y_k y'_k/\lambda t + (y/\lambda t)^2 pair_{k,k}$

      $LocalSet \leftarrow LocalSet \bigcup \{(idx_k, x_k, y_k, \frac{1}{\lambda t' s})\}$

      for $l = k + 1, \ldots, r$ **do**

        $y'_l \leftarrow y'_l + \frac{y_k}{\lambda t'} \cdot pair_{k,l}$

      if $norm > 1/\lambda$ **then**

        $s \leftarrow \frac{s}{\sqrt{\lambda \cdot norm}}$; $norm \leftarrow 1/\lambda$

        for $l = k + 1, \ldots, r$ **do**

          $y'_l \leftarrow \frac{y'_l}{\sqrt{\lambda \cdot norm}}$

  $H \leftarrow H.multiput(LocalSet)$

return s, H

- The rest: $O(r^2)$

The total computation cost is:

$$\frac{T}{r} \cdot O(\frac{rmd}{p} + rp + \frac{r^2 d}{p} + r^2) = T \cdot O(\frac{(m+r)d}{p} + p + r))$$

Considering the fact that $r = O(m)$ and optimal $\lambda = O(1/m)$ and $T = O(1/\lambda\delta\epsilon)$, we have computational cost:

$$O((m^2 d/p + mp) \cdot \frac{1}{\delta\epsilon}) + O(mr/\delta\epsilon) \tag{1}$$

Compared with $O(m^2 d/\delta\epsilon)$ of single machine, there are two additional terms that are not ideally scalable with p. One of them even increases with p, though generally it should be much smaller than the major item.

2.3.2 Communication Cost

In each round:

- Line 3, taking r samples: shuffle $O(rd)$
- Line 3, broadcasting samples: 1-to-all $O(rdp)$
- Line 5, all reduce(with combiners) to get estimated responses: all-to-1 $O(rp)$
- line 6, collect to send all pairwise inner-product to driver: all-to-1 $O(r^2)$
- Line 21, updating parameters: 1-to-1 $O(rd)$

Summing $\frac{T}{r}$ rounds of update, we have total communication cost:

$$\frac{T}{r} \cdot O(rdp + r^2)$$

.

Since optimal $\lambda = O(1/m)$ and $T = O(1/\lambda\delta\epsilon)$, the total communication cost can also be denoted as:

$$O((dp + r)m/\delta\epsilon) \tag{2}$$

We notice that the total communication cost is also increasing with p, the number of cores.

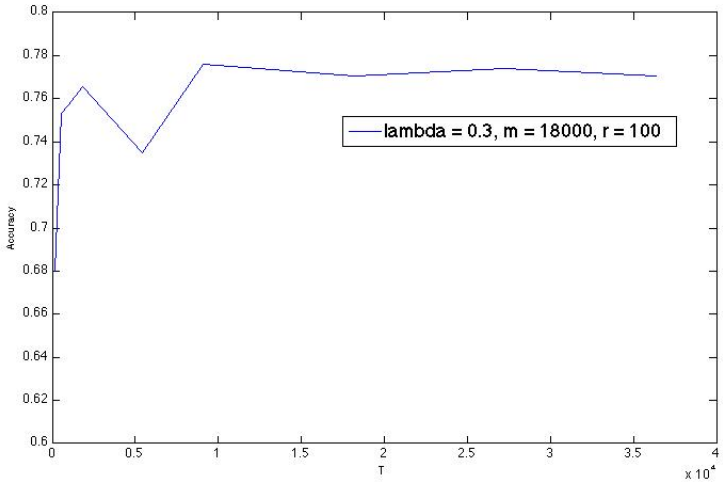

Figure 1: Test accuracy v.s number of iterations trained

3 Empirical Results

We mainly studied 3 aspects of the performance of our implementation. We are interested in the number of iterations costed before convergence, the affect of different packing size and the actual performance improved by adding more cores. All experiments utilize rbf kernel and the parameter γ for the kernel is set to 1.

3.1 Convergence

Number of iterations cost is a feature of algorithm itself. Studying this is also a good way to verify the correctness of our implementation. For this part we use the UCI Adult dataset from LibSVM website and take 18,000 data points as training set, and a separate 3000 data points test set. There are 123 features. We fix the packing size $r = 100$, and $\lambda = 0.3$, then try different number of iterations T and compute the corresponding test accuracy. The result is given by figure 1.

It turned out to converge after 10,000 iterations, which are about half the

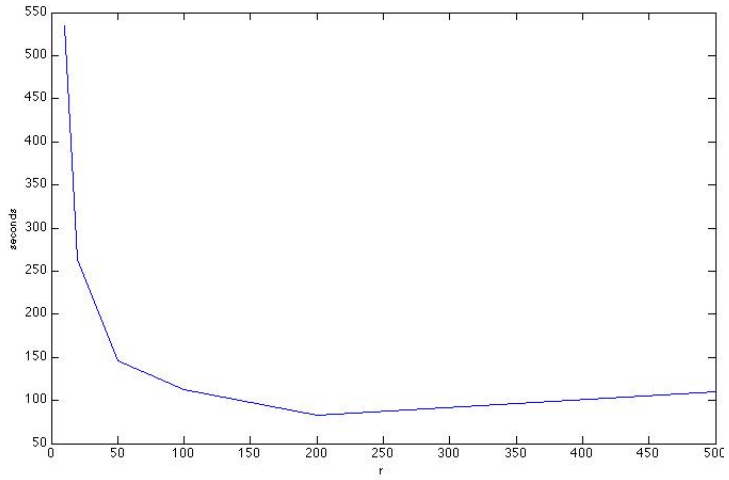

Figure 2: Runtime v.s r

data size. This coincides with $T = O(1/m)$ for choosing optimal regularization parameter.

3.2 Packing Size

From (1) and (2) we can see that increasing r will increase both computation and communication cost. However the number of communications would be less, so that we can save time cost on latency. Meanwhile the developer of IndexedRDD also suggests reducing the frequency of updating. Thus there should be some optimal r. We still use the Adult dataset, set r to different values and compare the time cost on training. The result is shown in figure 2.

It turns out that the optimal r for our setup is around 200. Notice that we are running on 4 cores locally, and we believe that level of parallelism, sample size or feature dimension may all affect the optimal choice of r.

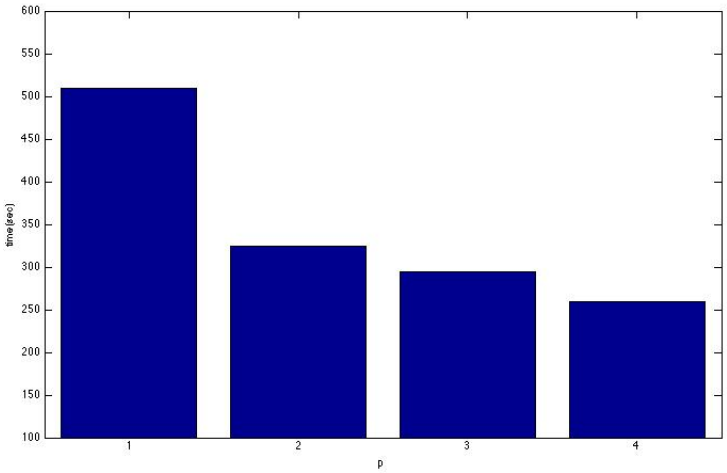

Figure 3: Runtime v.s p

3.3 Scalability

We are also interest in how the performance scales with number of processors used. In this part we use the real-sim dataset from LibSVM website, which has 57,925 training samples and 20,958 features. The training time, when using 1,2,3 or 4 cores, is shown in figure 3.

We can see that the computation does benefit from adding more cores. However ideally it should be better than this. One reason is that the IndexedRDD needs to be saved to disk every certain amount of updates (checkpoint), which is expensive and unscalable. We also believe that better scalability can be achieved by tuning packing size r for different p.

4 Conclusion

We studied and implemented a SGD algorithm that can solve Kernel SVM for large dataset and can benefit from parallelism, especially when the number of processors used are not too large. However the spark component we relies on, the IndexedRDD, still can be our bottleneck. Meanwhile, Some parameters

(i.e, T, r) need to be properly tuned for best performance.

References

[1] Zhu, Zeyuan Allen, et al, *"P-packSVM: Parallel primal gradient descent kernel SVM"*, *ICDM*, 2009.

[2] Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro, *"Pegasos: Primal Estimated sub-GrAdient SOlver for SVM"*, in *ICML*, 2007.