A Distributed Solver for Kernelized SVM

Haoming Li, Bangzheng He

Stanford ICME

haoming@stanford.edu
bzhe@stanford.edu

June 3, 2015

Overview

Support Vector Machines

- A widely used supervised learning model, originally for binary classification.
- Model represented as the normal vector of a separating hyper-plane, w, and has convex objective (primal):

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

  It has the form 2-norm regularization + empirical loss. $\{(x_i, y_i)|x_i \in \mathbb{R}^d, y_i \in \{-1, 1\}\}_{i=1}^m$ is the training data.
- The lagrange dual function has the form:

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

Kernels

- The optimal w can be written as a superposition of training data:

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

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

- Therefore we need not specify the transformation $\phi(x_i)$ 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.

$$rbf : K(x_i, x_j) = exp(-\gamma \|x_i - x_j\|^2)$$

Stochastic Gradient Descent: S-pack Algorithm

- Based on stochastic gradient descent, the empirical loss can be approximated by the hinge loss on a single training sample.
- At iteration t, we randomly pick up a sample (x_i, y_i), then we have the sub-gradient:

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

- For faster learning rate, update the predictor w:

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

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

S-pack sequential algorithm

Input: λ, T, training data
1 **Initialize:** $H = \emptyset$, $s = 1$, $norm = 0$
2 **for** $t = 1, 2, \ldots T$ **do**
3 Randomly pick training sample (x, y)
4 $y' \leftarrow s \sum_{(x_i, \beta_i) \in H} \beta_i K(x_i, x)$
5 $s \leftarrow (1 - 1/t)s$
6 **if** $yy' < 1$ **then**
7 $norm \leftarrow norm + 2yy'/\lambda t + (y/\lambda t)^2 K(x, x)$
8 **if** *key x is found in H* **then**
9 $(x, \beta) \leftarrow (x, \beta + y/\lambda ts)$
10 **else**
11 $H \leftarrow H \bigcup (x, y/\lambda ts)$
12 **if** $norm > 1/\lambda$ **then**
13 $s \leftarrow s/\sqrt{\lambda norm}$
14 $norm \leftarrow 1/\lambda$
15 **return** s, H

Training data size m, number of features $d(x \in \mathbb{R}^d)$. Only line 4 take at most $O(md)$ time, other commands run in constant time.
And it was analyzed in Pegasos[Shalev-Shwartz, Shai, et al.] that it requires $T = O(1/\lambda\delta\epsilon)$ iterations to get $|f(w) - f(w*)| < \epsilon$ with at least $1 - \delta$ probability.

Total runtime $O(md/\lambda\delta\epsilon)$. Since the optimal $\lambda = O(1/m)$, total runtime $O(m^2 d/\delta\epsilon)$

Parallel P-pack Algorithm

Assumptions:

- The feature dimension d is not too large, that is, a small number of data points can fit in single machine.
- The data size m is large so that it should be distributed as RDD.
- The model, i.e., support vectors, are also distributed.
- Kernel function $K(x_1, x_2)$ can be computed in $O(d)$.

Idea: parallelize the computation in each iteration, in particular the computation of sub-gradient, in which the most costly part is to evaluate the response of the chosen sample based on the current model.

Parallel P-pack Algorithm

- If the current model is stored as RDD$[x, y, \alpha]$, then given any sample (x_i, y_i), we can easily compute its response in parallel by broadcasting - map - all reduce. Then send this response back to the driver and finish the computation of sub gradient locally on worker, which takes constant time in spite of m or d or anything.
- The tricky part is updating the model. RDDs usually don't serve well as distributed hash tables.
- Luckily found IndexedRDD: developed by AMPLab, using Long keys and can efficiently lookup/update elements in RDD by key.

Parallel P-pack Algorithm: packing strategy

Suggested by the author, a packing strategy might help. In each iteration:

- Pick r rather than 1 samples at a time.
- Compute sub-gradient with respect to these samples sequentially.
- update the model with r updates at a time.

Doesn't reduce the computational cost at all (even increases it a little bit), but is desirable in practice. (Latency and efficiency of IndexedRDD).

Parallel P-pack for Kernal SVM

Processor i (total number of processors p)
Input: λ, T, r, D (training data rdd)
Initialize: $H = IndexedRDD(D.map((x, y) => (idx, x, y, alpha = 0))$, $s = 1$, $norm = 0$
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**
    $y'_k \leftarrow H.map(h => h.y * h.\alpha * K(x_k, h.x)).reduce(+)$
  Calculate $pair_{u,v} \leftarrow K(x_u, x_v)$ $(u = 1, \ldots, r\ v = 1, \ldots, r\ u \leq v)$ in distributed manner
  $LocalSet \leftarrow \emptyset$
  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**
      $norm \leftarrow norm + 2y_k y'_k / \lambda t + (y/\lambda t)^2 pair_{k,k}$
      $LocalSet \leftarrow LocalSet \bigcup \left\{(idx_k, x_k, y_k, \frac{1}{\lambda t' s})\right\}$
      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**
          $y'_l \leftarrow \frac{y'_l}{\sqrt{\lambda \cdot norm}}$
  Update H according to $LocalSet$
return s, H

Computational Cost

Data size m, feature dimension d, regularization parameter λ, packing size r, running T iterations (T/r rounds of updates), using p processors. In each round:

- Line 6, computing contribution to responses by each support vector for r samples: $O(rd \cdot \frac{m}{p})$
- Line 6, sum up responses from all processors: $O(rp)$
- Line 9, computing pairwise inner-products: $O(\frac{r^2 d}{p})$
- 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:

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

Compared with $O(m^2 d/\delta\epsilon)$ of single machine.

Communication Cost

Data size m, feature dimension d, regularization parameter λ, packing size r, running T iterations (T/r rounds of updates), using p processors. In each round:

- Line 4, taking r samples: shuffle $O(rd)$
- Line 4, broadcasting samples: 1-to-all $O(rdp)$
- Line 6, all reduce(with combiners) to get estimated responses: all-to-1 $O(rp)$
- line 7, collect to send all pairwise inner-product to driver: all-to-1 $O(r^2)$
- Line 23, 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:

$$O((dp + r)m/\delta\epsilon)$$

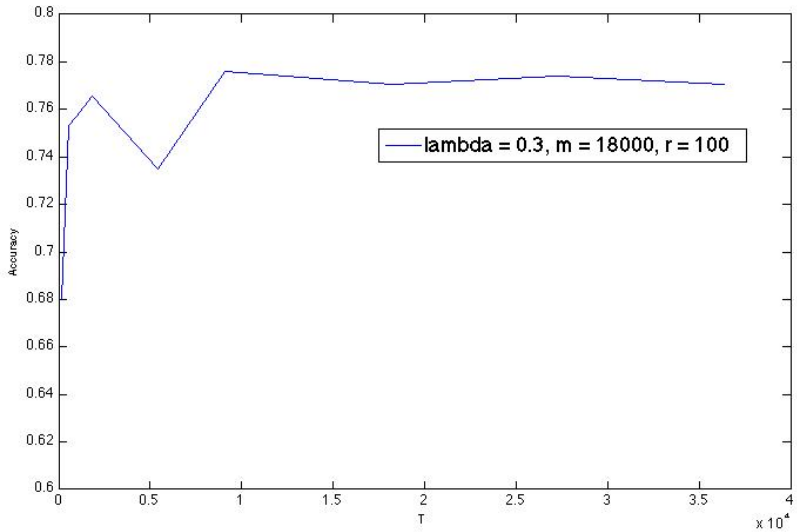
Empirical results

Using the "UCI Adult" dataset on LibSVM website. 18,000 training samples with 123 features. Rbf kernel is used and we study:

- Convergence with respect to number of iterations T.
- Performance with respect to packing size r.
- Scalability with respect to number of processors p.

Convergence

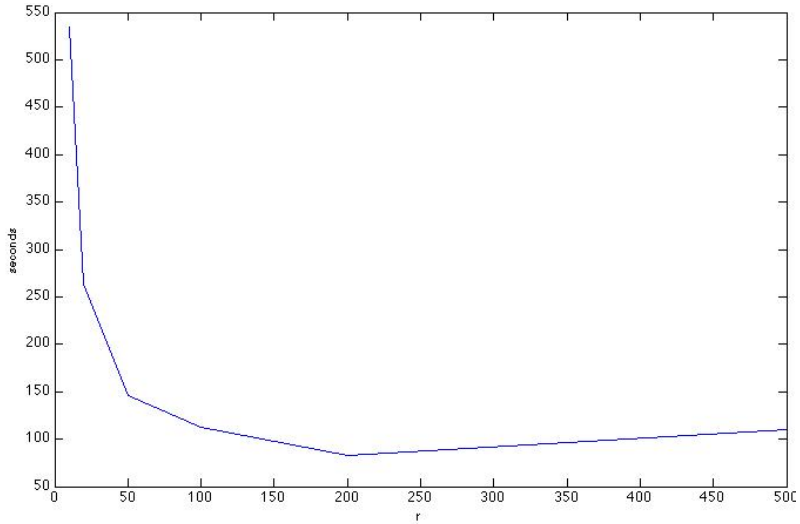
Study convergence: does this algorithm converge at all? How many iterations does it take? Test accuracy against number of iterations:



We find that this converges after 10000 iterations, about half of the data size m.

Packing size

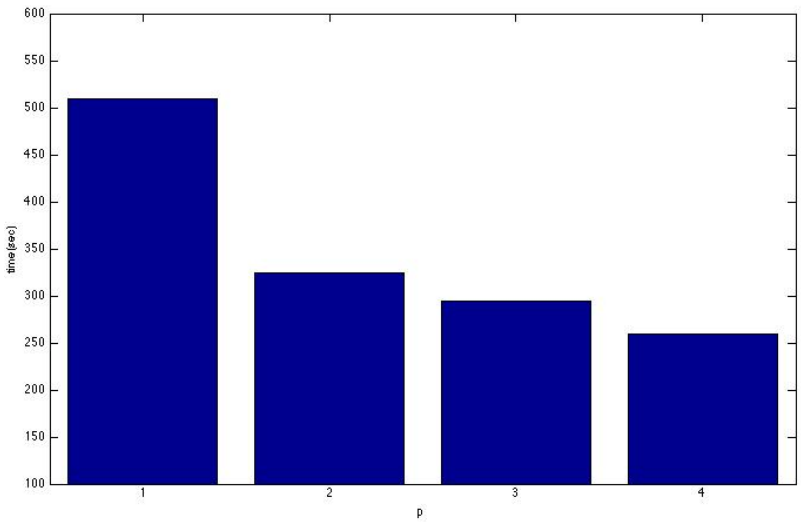
Study packing size: given different packing size r, study the runtime:



Empirically the optimal r in our case is around 200.

Scalability

Since we can only run on our laptops, only $p = 1$ to 4 are tried.



We do benefit from adding cores. We believe better scalability can be achieved if tuning packing size r for different p.

Conclusions

- We studied and implemented a SGD algorithm that can solve Kernel SVM and can benefit from adding more machines, especially when there are not already too many machines.
- Some parameters (i.e, T, r) need to be properly set for best performance.

References

Zhu, Zeyuan Allen, et al (2009)

P-packSVM: Parallel primal gradient descent kernel SVM.

Data Mining, 2009. ICDM'09 .

Shalev-Shwartz, Shai, et al. (2011)

Pegasos: Primal estimated sub-gradient solver for svm.

Mathematical programming 127.1 (2011): 3-30.

Thank you!