CS246: Mining Massive Datasets

Assignment number: 4

Submission time: ________ **and date:** ________

Fill in and include this cover sheet with each of your assignments. It is an honor code violation to write down the wrong time. Assignments are due at 9:30 am, either handed in at the beginning of class or left in the submission box on the 1^{st} floor of the Gates building, near the east entrance. Failure to include the coversheet with you assignment will be penalized by 2 points.

Each student will have a total of *two* free late periods. *One late period expires at the start of each class.* (Assignments are due on Thursdays, which means the first late period expires on the following Tuesday at 9:30am.) Once these late periods are exhausted, any assignments turned in late will be penalized 50% per late period. However, no assignment will be accepted more than *one* late period after its due date. (If an assignment is due to Thursday then we will not accept it after the following Thursday.)

Your name: Blaž Sovdat

Email: blaz.sovdat@gmail.com **SUNet ID:**

Collaborators:

I acknowledge and accept the Honor Code.

(Signed)

(For CS246 staff only)

Late days: 0 1

Section	Score
1	
2	
3	
4	
Total	

Comments: /

1 Support Vector Machine

(a) We give a sample training set of 5 points $\mathbf{x}_i \in \mathbb{R}^2$ with their respective classes $y_i \in \{-1, 1\}$ such that the set if infeasible under hard constraint SVM, but feasible under soft margin SVM:

- $(0, 1), -,$
- $(4, 0), -,$
- $(2, 2), +,$
- $(5, 1), +,$
- $(3, 3), -.$

This data set is clearly infeasible under hard constraint SVM as it is not linearly seperable. However, it is feasible under soft margin SVM. A concrete but trivial solution, assuming $C = 0$, is $\mathbf{w} := \mathbf{0}$ and $b := \xi_1 := \ldots \xi_n := 0$ (we can set b and ξ_i's to anything). (More generally, for any $\mathbf{w}$, b, and C the problem is feasible under soft margin SVM, because setting $\xi_i := \max(0, 1 - y_i(\mathbf{w}\mathbf{x}_i + b))$ satisfies all inequalities.)

(b) Let $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$ be an arbitrary dataset and suppose we are doing soft margin SVM. Note that setting ith slack to $\xi_i := \max(0, 1 - y_i(\mathbf{w}\mathbf{x}_i + b))$ for all $1 \leq i \leq n$ makes all the inequalities hold, where $\mathbf{w}$ and b are arbitrary. Thus the problem is feasible.

(c) Let $\{(\mathbf{x}_i, y_i) \mid 1 \leq i \leq n\}$ be our data set and let $(\mathbf{w}, b, \xi_1, \ldots, \xi_n)$ be feasible parameter set. Clearly, if $y_i(\mathbf{w}\mathbf{x}_i + b) \leq 0$ for arbitrary example, i.e., ith example is misclassified, then the corresponding slack will be $\xi_i \geq 1$, so each missclassification contributes at least 1 to the sum $\xi_1 + \xi_2 + \ldots + \xi_n$. This means $\sum_{i=1}^{n} \xi_i$ is an upper bound on the number of missclasifications.

(d) We compute $\nabla_b f(\mathbf{w}, b)$ for the batch gradient descent:

$$\nabla_b f(\mathbf{w}, b) = \frac{\partial f}{\partial b}(\mathbf{w}, b) = C \sum_{i=1}^{n} \frac{\partial L}{\partial b} L(\mathbf{x}_i, y_i),$$

where L is the hinge function with the partial derivative of

$$\frac{\partial L}{\partial b}(\mathbf{x}_i, y_i) = \begin{cases} 0 & y_i(\mathbf{w}\mathbf{x}_i + b) \geq 1, \\ -y_i & \text{otherwise.} \end{cases} \tag{1}$$

For stohastic gradient descent we use

$$\nabla_b f_i(\mathbf{w}, b) = C \frac{\partial L}{\partial b}(\mathbf{x}_i, y_i).$$

For mini batch gradient descent we have

$$\nabla_b f_\ell(\mathbf{w}, b) = C \sum_{i=bs \cdot \ell + 1}^{\min(n, (\ell+1)bs)} \frac{\partial L}{\partial b}(\mathbf{x}_i, y_i).$$

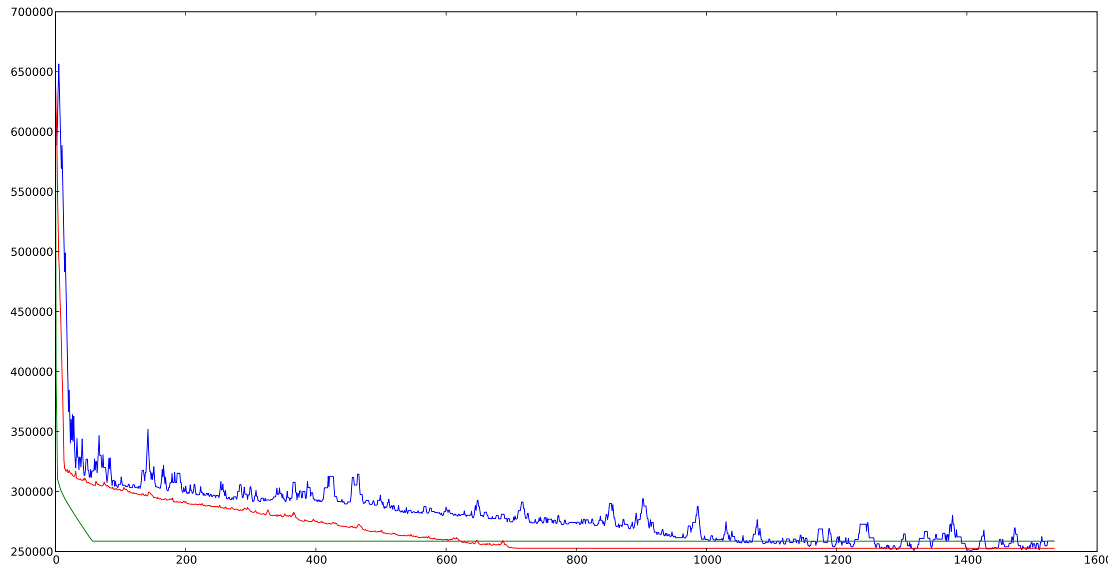

Figure 1: Value of $f_k(\mathbf{w}, b)$ at each iteration k for batch GD (green), stohastic GD (blue), and mini batch GD (red), where $k = 0, 1, \ldots, 1534$. See text for detailed explanation.

(e) See figure 1 for $f_k(\mathbf{w}, b)$ plots. (Parameters of descents are as in HW4Q1 text.) Batch GD converged (green line) in 57th iteration, mini batch GD (red line) in 713th, while stohastic GD converged (blue line) in 1534th iteration. In figure 1 we plotted $f_{57}(\mathbf{w}, n)$ instead of $f_i(\mathbf{w}, b)$ for all $i > 57$ for batch GD; we did similar thing for mini batch GD for $i > 713$ (this is why green and red line are "straight" from some point on). Code for batch GD is in listing 1, code for stohastic GD in listing 2, and code for mini batch GD in listing 3. (Source code is in the appendix at the end of the report.) The running times are in table 1; we computed running times using `timeit` Python module.

We see that stohastic GD takes least time per iteration but takes many iterations to converge; still, its running time is much lower than batch GD's. Batch GD takes lots of time per iteration as it needs to go through the whole dataset to evaluate the gradient; although it converges very quickly, it is still much slower than stohastic GD. Mini batch GD falls somewhere in between: it converges faster than stohastic GD and takes slightly more time per iteration; in our case it converged faster than both batch and stohastic GD. All variants converge to roughly the same objective function value.

(f) See listing 5 for code for this task. Figure 2 shows plot of percentage error — the fraction of missclassified examples on the test set — of stohastic GD (with parameters as in HW4Q1 text) as a function of regularization parameter $C = 1, 10, 50, 100, 200, 300, 400, 500$. We see that increasing C (this means we increase punishment for missclassifications) reduces percentage error on the test set. (If we increased C too much, we would overfit:

Method	Time
Batch GD	2125.11 seconds ($\approx$ 35.4 minutes)
Stohastic GD	508.17 seconds ($\approx$ 8.75 minutes)
Mini batch GD	326.584 seconds ($\approx$ 5.4 minutes)

Table 1: Running times for batch, stohastic, and mini batch GD implementations of SVM.

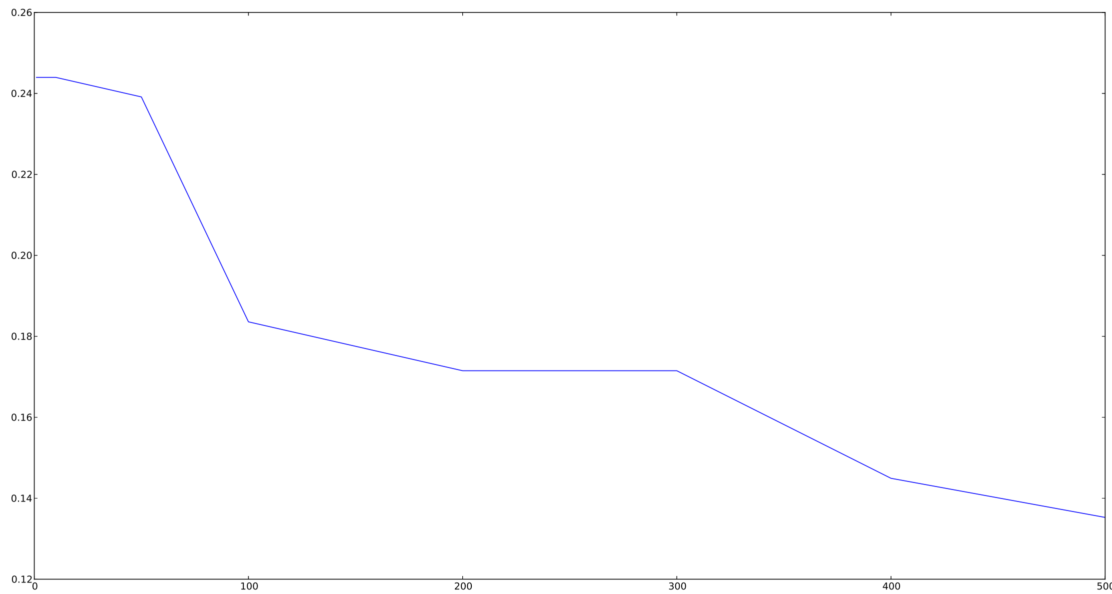

Figure 2: Plot of the percentage error of stohastic GD (parameters are in HW4Q1 text) as a function of regularization parameter C for $C = 1, 10, 50, 100, 200, 300, 400, 500$.

for very large C we practically ignore $\|\mathbf{w}\|^2/2$ factor. Similarly, sending $C \to 0$ ignores the data.) Percentage error decreases, then is "stable" around $C = 200, 300$ and then decreases some more.

2 Decision Tree Learning

(a) We compute $I(D)$ for each of the binary attributes. First note that we have $I(D) = 100(1 - (60/100)^2 - (40/100)^2) = 48$.

- For "likes wine" attribute $|D_L| = |D_R| = 50$ and $I(D_L) = I(D_R) = 50(1 - (30/50)^2 - (20/50)^2) = 24$, which gives us $I(D) - I(D_L) - I(D_R) = 0$.
- For "likes running" attribute we have $I(D_L) = 30(1 - (20/30)^2 - (10/30)^2) = 13.3$ and $I(D_R) = 70(1 - (40/70)^2 - (30/70)^2) = 34.29$, giving us $I(D) - I(D_L) - I(D_R) =$

0.38.

- For “likes pizza” attribute we have $I(D_L) = 80(1-(50/80)^2-(30/80)^2) = 37.5$ and $I(D_R) = 20(1-(10/20)^2-(10/20)^2) = 10$, giving us $I(D) - I(D_L) - I(D_R) = 0.5$.

We would use “likes pizza” attribute to split the root, because it has highest value of the Gini index based metric G. Roughly speaking, this means it alone (in the sense that we do not take into account attribute interactions; we just measure how well each single attribute classifies the data set) best classifies the data set.

(b) Under reasonable attribute estimation measures — like information gain and Gini index — the learner will identify a_1 as the most important attribute and use it as the split in the root node; all other attributes will apear in “lower layers” of the tree; the complete binary decision tree overfits the data.

It is obvious that the desired tree is the one with a single split on a_1, with leaf in the left branch ($a_1 = 1$) predicting 1 and leaf in the right branch ($a_1 = 0$) predicting 0. The reason for this is that simple models tend to generalize well; they are also easier to understand. (Very roughly, Occam's razor tells us we should favor simple hypotheses over complicated ones.)

(c) We use the following criterion for prunning:

$$\alpha = \frac{\text{error}(\text{pruned}(T,t),S) - \text{error}(T,S)}{|\,\text{nodes}(T)| - |\,\text{nodes}(\text{pruned}(T,t))|},$$

where $\text{error}(T,S)$ is error rate of T over the sample S. See figure 3 for original tree; figure 4 for T_1; figure 5 for T_2; figure 6 for T_3; and figure 7 for the final tree. Error rates (computed on the data from HW4Q2 figure 1) are included in the figure captions.

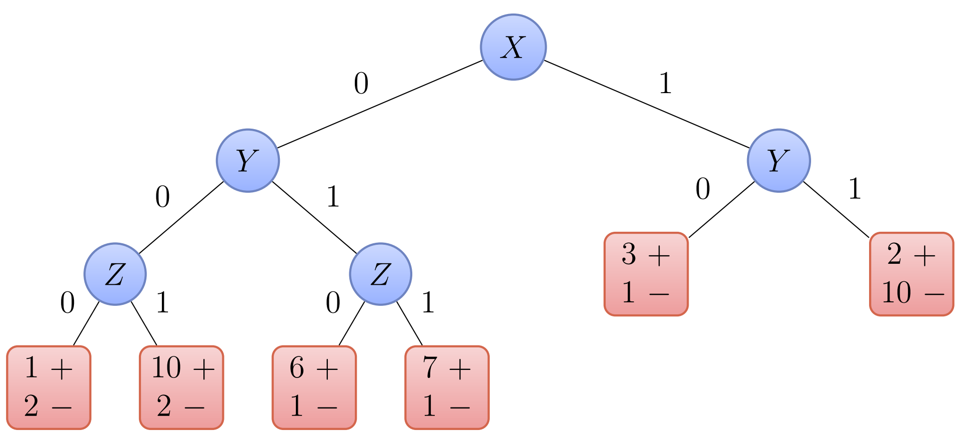

Figure 3: Decision tree T_0 with $err = (1+2+1+1+1+2)/46$.

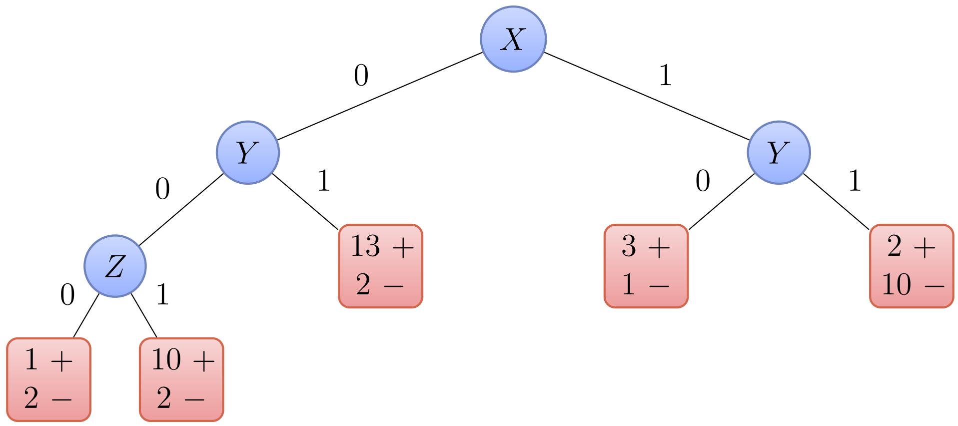

Figure 4: Decision tree T_1 with $err = (1 + 2 + 2 + 1 + 2)/46$.

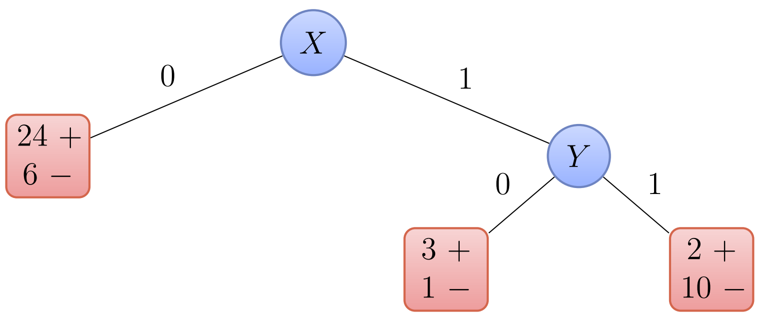

Figure 5: Decision tree T_2 with $err = (6 + 1 + 2)/46$.

(d) We now compute generalization errors for trees from figures 4–7 on the test data. We found that T_1 and T_3 have generalization error of $1/2$; tree T_2 has smaller generalization error of $1/4$; tree T_3 has zero generalization error; trivial tree T_4 has $1/2$ generalization error. Thus tree T_3 from figure 6 has the smallest generalization error.

3 Clustering Data Streams

We first remember the notation. Let $d : \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}^+$ be given by $d(x, y) := \|x - y\|_2$ and for $x \in \mathbb{R}^p$ and $T \subset \mathbb{R}^p$ we let

$$d(x, T) := \min_{z \in T} d(x, z).$$

For subsets $S, T \subset \mathbb{R}^p$ and a weight function $w : S \to \mathbb{R}^+$, define

$$\mathrm{cost}_w(S, T) := \sum_{x \in S} w(x) d(x, T)^2.$$

Also, let $T^\star := \arg\min_{|T|=k} \mathrm{cost}_w(S, T)$. We now turn to problems.

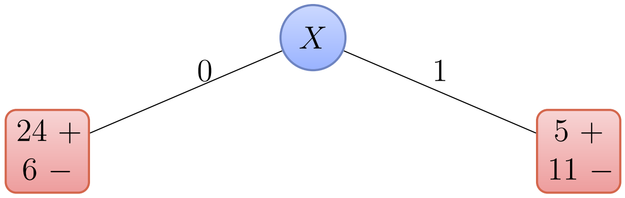

Figure 6: Decision tree T_3 with $err = (6+5)/46$.

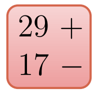

Figure 7: Decision tree T_4 with $err = 17/46$.

(a) Let $(a, b) \in \mathbb{R}_+$. Clearly we have

$$2a^2 + 2b^2 - (a+b)^2 = a^2 + b^2 - 2ab = (a-b)^2 \geq 0,$$

giving us $(a+b)^2 \leq 2a^2 + 2b^2$.

(b) We now show

$$\text{cost}(S, T) \leq 2\,\text{cost}_w(\hat{S}, T) + 2\sum_{i=1}^{\ell} \text{cost}(S_i, T_i).$$

First note that $\sum_{t_{ij} \in \hat{S}} |S_{ij}| = |S|$ and that each $x \in S$ belongs to exactly one S_i and exactly one S_{ij}. Pick arbitrary $x \in S$ and suppose $x \in S_i$ with $z := \arg\min_{z \in T} d(x, z)$. Now apply triangle inequality to get $d(x, z) \leq d(x, t_{ix}) + d(t_{ix}, z)$ for $t_{ix} := \arg\min_{t_{ix} \in T_i} d(x, t_{ix})$. Taking squares we get

$$\begin{aligned} d(x, z)^2 &\leq (d(x, t_{ix}) + d(t_{ix}, z))^2 \\ &\leq 2d(x, t_{ix})^2 + 2d(t_{ix}, z)^2, \end{aligned}$$

with the last inequality following by previous bullet. If $x \in S_i$ "belongs to" center $t_{ix} \in T_i$, then $\arg\min_{z \in T} d(x, z) = \arg\min_{z \in T} d(t_{ix}, z)$, so the proof is practically finished. We now sum over all elements and get what we want:

$$\begin{aligned} \sum_{x \in S} \text{cost}(x, T) &= \sum_{i=1}^{\ell} \sum_{x \in S_i} \min_{z \in T}\; d(x, z)^2 \\ &\leq \sum_{i=1}^{\ell} \sum_{x \in S_i} \left(2 \min_{t_{ix} \in T_i}\; d(x, t_{tix})^2 + 2 \min_{z \in T}\; d(t_{ix}, z)^2 \right) \\ &= 2\,\text{cost}_w(\hat{S}, T) + 2\sum_{i=1}^{\ell} \text{cost}(S_i, T_i). \end{aligned}$$

(c) We now prove $\sum_{i=1}^{\ell} \mathrm{cost}(S_i, T_i) \le \alpha \,\mathrm{cost}(S, T^\star)$. Note that $S_1, S_2, \dots, S_\ell$ form partition of S and by ALG's design $\mathrm{cost}(S_i, T_i) \le \alpha \,\mathrm{cost}(S_i, T')$ for all $|T'| = k$. We thus have

$$\begin{aligned}
\sum_{i=1}^{\ell} \mathrm{cost}(S_i, T_i) &\le \alpha \sum_{i=1}^{\ell} \mathrm{cost}(S_i, T^\star) \\
&= \alpha \sum_{i=1}^{\ell} \sum_{x \in S_i} d(x, T^\star)^2 \\
&= \alpha \sum_{x \in S} d(x, T^\star)^2 \\
&= \alpha \,\mathrm{cost}(S, T^\star).
\end{aligned}$$

(d) We now show $\mathrm{cost}_w(\hat{S}, T) \le \alpha \,\mathrm{cost}_w(\hat{S}, T^\star)$. Let $T_\star := \underset{|T'|=k}{\arg\min}\ \mathrm{cost}_w(\hat{S}, T')$, so $\mathrm{cost}_w(\hat{S}, T) \le \alpha \,\mathrm{cost}_w(\hat{S}, T_\star)$. Now since $\mathrm{cost}_w(\hat{S}, T^\star) \ge \mathrm{cost}_w(\hat{S}, T_\star)$, we put all together and get

$$\alpha \,\mathrm{cost}_w(\hat{S}, T^\star) \ge \alpha \,\mathrm{cost}_w(\hat{S}, T_\star) \ge \mathrm{cost}_w(\hat{S}, T),$$

so we "proved" $\mathrm{cost}_w(\hat{S}, T) \le \alpha \,\mathrm{cost}_w(\hat{S}, T^\star)$.

(e) In this bullet we prove $\mathrm{cost}_w(\hat{S}, T^\star) \le 2 \sum_{i=1}^{\ell} \mathrm{cost}(S_i, T_i) + 2 \,\mathrm{cost}(S, T^\star)$, using similar arguments as in (b). Fix $t_{ij} \in \hat{S}$. (Recall that each center t_{ij} has associated $S_{ij} \subseteq S_i$.) Then for each $x \in S_{ij}$ we have $d(t_{ij}, z) \le d(t_{ij}, x) + d(x, z)$ for $z = \arg\min_{z \in T^\star} d(t_{ij}, z) = d(x, T^\star)$. Using (a) we get $d(t_{ij}, z)^2 \le 2d(t_{ij}, x)^2 + 2d(x, z)^2$. We now sum over all elements to get the desired inequality:

$$\begin{aligned}
\mathrm{cost}_w(\hat{S}, T^\star) &= \sum_{t_{ij} \in \hat{S}} |S_{ij}| d(t_{ij}, T^\star)^2 \\
&\le \sum_{t_{ij} \in \hat{S}} \sum_{x \in S_{ij}} \left(2d(t_{ij}, x)^2 + 2d(x, T^\star)^2 \right) \\
&= 2 \sum_{i=1}^{\ell} \sum_{x \in S_i} d(t_{ij}, x)^2 + 2 \sum_{x \in S} d(x, T^\star)^2 \\
&= 2 \sum_{i=1}^{\ell} \mathrm{cost}(S_i, T_i) + 2 \,\mathrm{cost}(S, T^\star)
\end{aligned}$$

(Above we are just summing over different partitions of the same set and rearranging the summation terms, so everything is fine.)

(f) Using previous inequalities we have:

$$\begin{aligned}
\operatorname{cost}(S,T) &\le 2\operatorname{cost}_w(\hat{S},T) + 2\sum_{i=1}^{\ell}\operatorname{cost}(S_i,T_i) \\
&\le 2\alpha\operatorname{cost}_w(\hat{S},T^\star) + 2\alpha\operatorname{cost}(S,T^\star) \\
&\le 2\alpha\left(2\sum_{i=1}^{\ell}\operatorname{cost}(S,T_i) + 2\operatorname{cost}(S,T^\star)\right) + 2\alpha\operatorname{cost}(S,T^\star) \\
&\le 2\alpha\left(2\alpha\operatorname{cost}(S,T^\star) + 2\operatorname{cost}(S,T^\star)\right) + 2\alpha\operatorname{cost}(S,T^\star) \\
&= (4\alpha^2 + 6\alpha)\operatorname{cost}(S,T^\star).
\end{aligned}$$

(g) We show that we with a good choice of partitioning, ALGSTR works with $O(\sqrt{nk})$ memory. If we pick $\ell = O(\sqrt{n/k})$ then we use $n/\ell = O(\sqrt{nk})$ memory in each of ℓ steps and $k\ell = O(\sqrt{nk})$ memory for the final clustering. We thus use $O(\sqrt{nk})$ memory in total.

4 Data Streams

Let $S = \langle a_1, a_2, \dots, a_t\rangle$ be a data stream of items from $\{1,2,\dots,n\}$. Assume for any $1 \le i \le n$, $F[i]$ is the number of times item i has appeared in S. For given parameters $\epsilon > 0$ and $\delta > 0$ the algorithm picks $\lceil \log(1/\delta)\rceil$ independent hash functions $h_j : \{1,\dots,n\} \to \{1,2,\dots,\lceil e/\epsilon\rceil\}$. Let $\widetilde{F}[i] := \min_j c_{j,h_j(i)}$. (Note: Our notation here is somewhat cumbersome; sorry for inconvinience.)

(a) We claim $\widetilde{F}[i] \ge F[i]$ for all $1 \le i \le n$. To see this note that if item i appeared $F[i]$ times, then $c_{j,h_j(i)} \ge F[i]$ for all $1 \le j \le n$, because each hash function h_j will increment the count $c_{j,h_j(i)}$ at each of the $F[i]$ appearances of i. (Count may be greater, depending on the number of collisions.)

(b) For any $1 \le i \le n$ and $1 \le j \le \lceil \log(1/\delta)\rceil$ we will prove

$$\mathbb{E}[c_{j,h_j(i)}] \le F[i] + \frac{\epsilon}{e}(t - F[i]).$$

First fix item $i \in \{1,2,\dots,n\}$ and define random varaible $X_j := c_{j,h_j(i)}$. We already know $X_j \ge F[i]$; we just need to bound "surplus"[1] in our count. To do this, we define indicator random variable $Y_{j,k} := 1$ if $h_j(i) = h_k(i)$ for $k \ne i$ and $k \in \{1,2,\dots,n\}$ and $Y_{j,k} := 0$ otherwise. (Intuitively, sum of $Y_{j,k}$'s counts collisions, and thus surplus in our counts.) We can now write X_j in terms of $F[i]$ and indicator random variables:

$$X_j = F[i] + \sum_{1 \le k \ne i \le n} Y_{j,k} F[k].$$

[1] This is translation of "presežek".

Note $\mathbb{E}[Y_{j,k}] = \Pr[Y_{j,k} = 1] = \Pr[h_j(k) = h_j(i)] \le \epsilon/e$ because hash functions are uniform and independent. By linearity of expcetation we have

$$\begin{aligned}
\mathbb{E}[X_j] &= F[i] + \sum_{1 \le k \ne i \le n} \mathbb{E}[Y_{j,k}] F[k] \\
&\le F[i] + \frac{\epsilon}{e} \sum_{1 \le k \ne i \le n} F[k] \\
&= F[i] + \frac{\epsilon}{e}(t - F[i]),
\end{aligned}$$

establishing the inequality, because we fixed arbitrary item i. (We used $t = \sum_i F[i]$ and $\sum_{k \ne i} = t - F[i]$.)

(c) We now show

$$\Pr[\widetilde{F}[i] \le F[i] + \epsilon t] \ge 1 - \delta.$$

Recall that the Markov's inequality says $\Pr[X \ge a] \le \mathbb{E}[X]/a$ for real constant $a > 0$ and nonnegative random variable X. For X_j from previous bullet we have $\Pr[X_j - F[i] \ge \epsilon t] \le \mathbb{E}[X_i - F[i]]/(t\epsilon) \le 1/e$. First note that $\Pr[\widetilde{F}[i] \le F[i] + \epsilon t] = \Pr[\widetilde{F}[i] - F[i] \le \epsilon t]$ and (because $\widetilde{F}[i] = \min_j \; c_{j,h_j(i)}$) that $\widetilde{F}[i] - F[i] \le \epsilon t$ implies $c_{j,h_j(i)} - F[i] \le \epsilon t$ for all $1 \le j \le n$. We will "work with" event '$\widetilde{F}[i] \ge F[i] + \epsilon t$'. Thus

$$\begin{aligned}
\Pr[\widetilde{F}[i] \ge F[i] + \epsilon t] &= \Pr[\widetilde{F}[i] - F[i] \ge \epsilon t] \\
&= \Pr\left[\bigwedge_{j=1}^{\lceil \log(1/\delta) \rceil} \left(c_{j,h_j(i)} - F[i] \ge \epsilon t \right) \right] \\
&= \prod_{j=1}^{\lceil \log(1/\delta) \rceil} \Pr[c_{j,h_j(i)} - F[i] \ge \epsilon t] \\
&\le (1/e)^{\lceil \log(1/\delta) \rceil} \le (1/e)^{\log(1/\delta)} = (1/e)^{-\log \delta} = e^{\log \delta} = \delta,
\end{aligned}$$

because we used natural logarithms for this question, i.e., $\log x := \log_e x$. Finally, note that $\Pr[\widetilde{F}[i] \le F[i] + \epsilon t] = 1 - \Pr[\widetilde{F}[i] \ge F[i] + \epsilon t] \ge 1 - \delta$. This finishes the proof.

(d) In HW3Q4 we had a streaming implementation of algorithm for finding dense communities in networks, where we processed the input graph edge-by-edge and counted degrees of each of the nodes; it uses at least $4|V| = O(|V|)$ bytes of memory for counting vertex degrees, assuming counters are 4B integers. We could use algorithm from this question to estimate (two times) the degree of each vertex with $\epsilon > 0$ parameter of HW3Q4 algorithm, because $2\deg_G(v)$ equals the number of edges that v "touches". Setting, for instance, $\delta := 1/|V|^2$ we would this way use only $O(\log |V|)$ space (because ϵ is just some constant) for vertex degree counts. The total asymptotic space complexity of the HW3Q4 algorithm is unchanged.

A Source code for SVM

This appendix contains source code we wrote for the first question. (There is lots of debugging output; please ignore it.)

A.1 SVM via batch GD, stohastic GD, and mini batch GD

Listing 1: SVM via batch gradient descent.

```
#!/usr/bin/python

import os, sys, time
import numpy as np
from numpy import linalg
import timeit

# derivative of hinge by wj
def hinge(X, Y, w, b, j):
  return 100*sum([0 if Y[i]*(np.dot(X[i], w)+b) >= 1 else -X[i][j]*Y[i] for i in range(len(X))]) # C=100

# compute cost function --- iterates over the whole dataset
def f(X, Y, w, b):
  return linalg.norm(w)**2/2 + 100*sum([max(0, 1.0-Y[i]*(np.dot(w, X[i])+b)) for i in range(len(X))]) # C=100

# derivative of f by b
def grad_b(X, Y, w, b):
  return 100*sum([0 if Y[i]*(np.dot(X[i], w)+b) >= 1 else -Y[i] for i in range(len(X))]) # C=100

# batch gradient descent
def bgd(X, Y, w, b, ni = 0.0000003, eps = 0.25):
  assert len(X) == len(Y), "Counts do not match"
  k = 0
  crr_f = prev_f = f(X, Y, w, b)
  delta_cost = 1.0
  print "crr_cost: ", crr_f
  while True:
    start = time.time()
    # do the update
    tw = list(w)
    for j in range(len(X[0])):
      tw[j] = w[j] - ni*(w[j] + hinge(X, Y, w, b, j))
    b = b - ni * grad_b(X, Y, w, b)
    w = list(tw)
    end = time.time()
    print "secs: ", end-start
    print "iter: ", k
    print "-"*80
```

```
    k = k+1
    crr_f = f(X, Y, w, b)
    print "crr_cost: ", crr_f
    delta_cost = (abs(prev_f - crr_f))*100/(prev_f)
    print "delta_cost: ", delta_cost
    if delta_cost < eps:
      print "Converged after", k, "steps!"
      break
    prev_f = crr_f
  print "w=", w
  print "b=", b
  return (w, b)

def load_train(fname):
  return [[int(el) for el in line.split(',')] for line in open(fname).read().split()]

def load_test(fname):
  return [int(el) for el in open(fname).read().split()]

# entry point
if __name__ == '__main__':
  #X = load_train('features.txt')
  #Y = load_test('target.txt')
  #w = [0 for x in range(len(X[0]))]
  #b = 0
  #(w, b) = bgd(X, Y, w, b)
  s = " X = load_train('features.txt')\n\
  Y = load_test('target.txt')\n\
  w = [0 for x in range(len(X[0]))]\n\
  b = 0\n\
  (w, b) = bgd(X, Y, w, b)"
  print(timeit.timeit(s, setup="from __main__ import *", number=1))
```

Listing 2: SVM via stohastic gradient descent.

```
#!/usr/bin/python

import os, sys, time
import numpy as np
from numpy import linalg
import random as rnd
import timeit

# derivative of hinge by wj; here, i is index of the training example
def hinge(X, Y, w, b, j, i):
  return 100*(0 if Y[i]*(np.dot(X[i], w)+b) >= 1 else -X[i][j]*Y[i]) # C=100

# compute cost function --- iterates over the whole dataset
def f(X, Y, w, b):
  return linalg.norm(w)**2/2 + 100*sum([max(0, 1.0-Y[i]*(np.dot(w, X[i])+b)) for i in range(len(X))]) # C=100
```

```

# derivative of f by b
def grad_b(X, Y, w, b, i):
  return 100*(0 if Y[i]*(np.dot(X[i], w)+b) >= 1 else -Y[i]) # C=100

# stohastic gradient descent
def sgd(X, Y, w, b, ni = 0.0001, eps = 0.001):
  I = range(len(X))
  rnd.shuffle(I)
  X = [X[i] for i in I]
  Y = [Y[i] for i in I]
  i, k = 1, 0
  n = len(X)
  crr_f = prev_f = f(X, Y, w, b)
  print "crr_cost: ", crr_f
  delta_cost = 0.0
  while True:
    tw = list(w)
    # update
    for j in range(len(X[0])):
      tw[j] = w[j] - ni*(w[j] + hinge(X, Y, w, b, j, i)) # derivative of hinge
          by w[j]
    b = b - ni*grad_b(X, Y, w, b, i)
    w = list(tw)
    i = (i % n) + 1
    k = k+1
    crr_f = f(X, Y, w, b)
    print "crr_cost: ", crr_f
    delta_cost = 0.5*delta_cost + 0.5*(abs(prev_f - crr_f))*100/(prev_f)
    print "k=", k
    print "delta_cost: ", delta_cost
    print "-"*80
    if delta_cost < eps:
      print "[DONE] Converged after", k, "steps"
      break
    prev_f = crr_f
  return (w, b)

def load_train(fname):
  return [[int(el) for el in line.split(',')] for line in open(fname).read().
      split()]

def load_test(fname):
  return [int(el) for el in open(fname).read().split()]

# entry point
if __name__ == '__main__':
  #X = load_train('features.txt')
  #Y = load_test('target.txt')
  #w = [0 for x in range(len(X[0]))]
  #b = 0
```

```
  #(w, b) = sgd(X, Y, w, b)
  s = " X = load_train('features.txt')\n\
  Y = load_test('target.txt')\n\
  w = [0 for x in range(len(X[0]))]\n\
  b = 0\n\
  (w, b) = sgd(X, Y, w, b)"
  print(timeit.timeit(s, setup="from __main__ import *", number=1))
```

Listing 3: SVM via mini batch gradient descent.

```
#!/usr/bin/python

import os, sys, time
import numpy as np
from numpy import linalg
import random as rnd
import timeit

# derivative of hinge by wj
def hinge(X, Y, w, b, j, l, bs):
  return 100*sum([0 if Y[i]*(np.dot(X[i], w)+b) >= 1 else -X[i][j]*Y[i] for i in range(l*bs+1, min(len(X), (l+1)*bs))]) # C=100

# compute cost function --- iterates over the whole dataset
def f(X, Y, w, b):
  return linalg.norm(w)**2/2 + 100*sum([max(0, 1.0-Y[i]*(np.dot(w, X[i])+b)) for i in range(len(X))]) # C=100

# derivative of f by b
def grad_b(X, Y, w, b, l, bs):
  return 100*sum([0 if Y[i]*(np.dot(X[i], w)+b) >= 1 else -Y[i] for i in range(l*bs+1, min(len(X), (l+1)*bs))]) # C=100

# mini batch gradient descent
def mini_bgd(X, Y, w, b, ni = 0.00001, eps = 0.01, bs = 20):
  I = range(len(X))
  rnd.shuffle(I)
  # reorder
  X = [X[i] for i in I]
  Y = [Y[i] for i in I]
  n = len(X)
  l, k = 0, 0
  crr_f = prev_f = f(X, Y, w, b)
  delta_cost = 0.0
  print "crr_cost: ", crr_f
  while True:
    start = time.time()
    # do the update
    tw = list(w)
    for j in range(len(X[0])):
      tw[j] = w[j] - ni*(w[j] + hinge(X, Y, w, b, j, l, bs))
    b = b - ni * grad_b(X, Y, w, b, l, bs)
```

```
    w = list(tw)
    end = time.time()
    print "secs: ", end-start
    print "iter: ", k
    print "-"*80
    k = k+1
    crr_f = f(X, Y, w, b)
    print "crr_cost: ", crr_f
    delta_cost = 0.5*delta_cost + 0.5*(abs(prev_f - crr_f))*100/(prev_f)
    print "delta_cost: ", delta_cost
    if delta_cost < eps:
      print "[DONE] Converged", k, "steps!"
      break
    l = (l + 1) % ((n + bs - 1)/bs)
    prev_f = crr_f
  print "w=", w
  print "b=", b
  return (w, b)

def load_train(fname):
  return [[int(el) for el in line.split(',')] for line in open(fname).read().split()]

def load_test(fname):
  return [int(el) for el in open(fname).read().split()]

# entry point
if __name__ == '__main__':
  #X = load_train('features.txt')
  #Y = load_test('target.txt')
  #w = [0 for x in range(len(X[0]))]
  #b = 0
  #(w, b) = mini_bgd(X, Y, w, b)
  s = " X = load_train('features.txt') \n\
  Y = load_test('target.txt') \n\
  w = [0 for x in range(len(X[0]))] \n\
  b = 0 \n\
  (w, b) = mini_bgd(X, Y, w, b)"
  print(timeit.timeit(s, setup="from __main__ import *", number=1))
```

Plots were generated by running `python hw4q1-bgd.py > hw4q1-bgd-a.txt` (similarly run mini batch GD and stohastic GD) and then running `plot.py` from listing 4.

Listing 4: Code snippet used for plotting.

```
#!/usr/bin/python

import os, sys, time
import numpy as np
from matplotlib import pyplot

if __name__ == '__main__':
```

```
# Batch gradient descent
Y_bgd = []
X = []
k = 0
for ln in open('hw4q1-bgd-a.txt').read().split('\n'):
  if ln[:8] == 'crr_cost':
    X.append(k)
    Y_bgd.append(float(ln.split()[1]))
    k = k+1
pyplot.plot(X, Y_bgd)
pyplot.show()
print "X = ", X
print "Y = ", Y_bgd

# Stohastic gradient descent
k = 0
X = []
Y_sgd = []
for ln in open('hw4q1-sgd-a.txt').read().split('\n'):
  if ln[:8] == 'crr_cost':
    X.append(k)
    Y_sgd.append(float(ln.split()[1]))
    k = k+1
for i in range(len(Y_sgd) - len(Y_bgd)): Y_bgd.append(Y_bgd[-1])
pyplot.plot(X, Y_sgd, X, Y_bgd)
pyplot.show()
print "X = ", X
print "Y = ", Y_sgd

# Mini Batch Gradient Descent
k = 0
#X = []
Y_mbgd = []
for ln in open('mini-bgd-a.txt').read().split('\n'):
  if ln[:8] == 'crr_cost':
    #X.append(k)
    Y_mbgd.append(float(ln.split()[1]))
    k = k+1
for i in range(len(Y_sgd) - len(Y_mbgd)): Y_mbgd.append(Y_mbgd[-1])
pyplot.plot(X, Y_sgd, X, Y_bgd, X, Y_mbgd)
pyplot.show()
print "X = ", X
print "Y = ", Y_mbgd
```

A.2 Code used to experiment with regularization

Listing 5: Code for HW4Q1 item (f).

```
#!/usr/bin/python

```

```python
import os, sys, time
import numpy as np
from numpy import linalg
import random as rnd
from matplotlib import pyplot

# derivative of hinge by wj; here, i is index of the training example
def hinge(X, Y, w, b, j, i, C):
  return C*(0 if Y[i]*(np.dot(X[i], w)+b) >= 1 else -X[i][j]*Y[i])

# compute cost function --- iterates over the whole dataset
def f(X, Y, w, b, C):
  return linalg.norm(w)**2/2 + C*sum([max(0, 1.0-Y[i]*(np.dot(w, X[i])+b)) for i in range(len(X))])

# derivative of f by b
def grad_b(X, Y, w, b, i, C):
  return C*(0 if Y[i]*(np.dot(X[i], w)+b) >= 1 else -Y[i]) # C=100

# stohastic gradient descent
def sgd(X, Y, w, b, ni = 0.0001, eps = 0.001, C = 100):
  I = range(len(X))
  rnd.shuffle(I)
  X = [X[i] for i in I]
  Y = [Y[i] for i in I]
  i, k = 1, 0
  n = len(X)
  crr_f = prev_f = f(X, Y, w, b, C)
  #print "crr_cost: ", crr_f
  delta_cost = 0.0
  while True:
    tw = list(w)
    # update
    for j in range(len(X[0])):
      tw[j] = w[j] - ni*(w[j] + hinge(X, Y, w, b, j, i, C)) # derivative of hinge by w[j]
    b = b - ni*grad_b(X, Y, w, b, i, C)
    w = list(tw)
    i = ((i+1) % n)
    k = k+1
    crr_f = f(X, Y, w, b, C)
    #print "crr_cost: ", crr_f
    delta_cost = 0.5*delta_cost + 0.5*(abs(prev_f - crr_f))*100/(prev_f)
    if k%500 == 0: print "k=", k
    #print "delta_cost: ", delta_cost
    #print "-"*80
    if delta_cost < eps:
      #print "[DONE] Converged after", k, "steps"
      break
    prev_f = crr_f
  print "Final f=", crr_f, "after", k, "steps"
```

```
  return (w, b)

def load_ftr(fname):
  return [[int(el) for el in line.split(',')] for line in open(fname).read().split()]

def load_trg(fname):
  return [int(el) for el in open(fname).read().split()]

def est_err(X, Y, w, b):
  n = len(X)
  miss = 0
  for i in range(n):
    if Y[i]*(np.dot(X[i], w)+b) <= 0: miss = miss+1
  print "[DEBUG] Misclassified", miss, "out of", n
  return 1.0*miss/n

# entry point
if __name__ == '__main__':
  Cs = [1, 10, 50, 100, 200, 300, 400, 500]
  # train data
  X = load_ftr('features.train.txt')
  Y = load_trg('target.train.txt')
  # test data
  X_test = load_ftr('features.test.txt')
  Y_test = load_trg('target.test.txt')
  E = []
  for C in Cs:
    X = load_ftr('features.train.txt')
    Y = load_trg('target.train.txt')
    w = [0 for x in range(len(X[0]))]
    b = 0
    (w, b) = sgd(X, Y, w, b, 0.0001, 0.001, C)
    print "w=", w
    print "b=", b
    E.append(est_err(X_test, Y_test, w, b))
    print "For C=", C, "the percent error equals", E[-1]
    print "-" * 80
  print "E=", E
  pyplot.plot(Cs, E)
  pyplot.show()
```

References