CS246: Mining Massive Data Sets **Winter 2020**

Problem Set 4

Please read the homework submission policies at http://cs246.stanford.edu.

1 Implementation of SVM via Gradient Descent (30 points)

Here, you will implement the soft margin SVM using different gradient descent methods as described in the section 12.3.4 of the textbook. To recap, to estimate the $\mathbf{w}, b$ of the soft margin SVM, we can minimize the cost:

$$f(\mathbf{w}, b) = \frac{1}{2}\sum_{j=1}^{d}(w^{(j)})^2 + C\sum_{i=1}^{n}\max\left\{0, 1 - y_i\left(\sum_{j=1}^{d} w^{(j)}x_i^{(j)} + b\right)\right\}. \quad (1)$$

In order to minimize the function, we first obtain the gradient with respect to $w^{(j)}$, the jth item in the vector $\mathbf{w}$, as follows.

$$\nabla_{w^{(j)}} f(\mathbf{w}, b) = \frac{\partial f(\mathbf{w}, b)}{\partial w^{(j)}} = w^{(j)} + C\sum_{i=1}^{n}\frac{\partial L(x_i, y_i)}{\partial w^{(j)}}, \quad (2)$$

where:

$$\frac{\partial L(x_i, y_i)}{\partial w^{(j)}} = \begin{cases} 0 & \text{if } y_i\,(\mathbf{x_i}\cdot\mathbf{w} + b) \geq 1 \\ -y_i x_i^{(j)} & \text{otherwise.} \end{cases}$$

Now, we will implement and compare the following gradient descent techniques:

1. **Batch gradient descent**: Iterate through the entire dataset and update the parameters as follows:

   k = 0
   while convergence criteria not reached **do**
     for $j = 1, ..., d$ **do**
       Update $w^{(j)} \leftarrow w^{(j)} - \eta\nabla_{w^{(j)}} f(\mathbf{w}, b)$
     end for
     Update $b \leftarrow b - \eta\nabla_b f(\mathbf{w}, b)$
     Update $k \leftarrow k + 1$
   end while

   where,
   n is the number of samples in the training data,
   d is the dimensions of $\mathbf{w}$,

η is the learning rate of the gradient descent, and
$\nabla_{w^{(j)}} f(\mathbf{w}, b)$ is the value computed from computing equation (2) above and $\nabla_b f(\mathbf{w}, b)$ is the value computed from your answer in question (a) below.

The *convergence criteria* for the above algorithm is $\Delta_{\%cost} < \epsilon$, where

$$\Delta_{\%cost} = \frac{|f_{k-1}(\mathbf{w}, b) - f_k(\mathbf{w}, b)| \times 100}{f_{k-1}(\mathbf{w}, b)}. \tag{3}$$

where,
$f_k(\mathbf{w}, b)$ is the value of equation (1) at kth iteration,
$\Delta_{\%cost}$ is computed at the end of each iteration of the while loop.
Initialize $\mathbf{w} = \mathbf{0}, b = 0$ and compute $f_0(\mathbf{w}, b)$ with these values.
For this method, use $\eta = 0.0000003, \epsilon = 0.25$

2. **Stochastic gradient descent**: Go through the dataset and update the parameters, one training sample at a time, as follows:

   Randomly shuffle the training data
   $i = 1, k = 0$
   while convergence criteria not reached **do**
     for $j = 1, ..., d$ **do**
       Update $w^{(j)} \leftarrow w^{(j)} - \eta \nabla_{w^{(j)}} f_i(\mathbf{w}, b)$
     end for
     Update $b \leftarrow b - \eta \nabla_b f_i(\mathbf{w}, b)$
     Update $i \leftarrow (i \mod n) + 1$
     Update $k \leftarrow k + 1$
   end while

   where,
   n is the number of samples in the training data,
   d is the dimensions of $\mathbf{w}$,
   η is the learning rate and
   $\nabla_{w^{(j)}} f_i(\mathbf{w}, b)$ is defined for a single training sample as follows:

$$\nabla_{w^{(j)}} f_i(\mathbf{w}, b) = \frac{\partial f_i(\mathbf{w}, b)}{\partial w^{(j)}} = w^{(j)} + C \frac{\partial L(x_i, y_i)}{\partial w^{(j)}}$$

   (Note that you will also have to derive $\nabla_b f_i(\mathbf{w}, b)$, but it should be similar to your solution to question (a) below.
   The *convergence criteria* here is $\Delta_{cost}^{(k)} < \epsilon$, where

$$\Delta_{cost}^{(k)} = 0.5 * \Delta_{cost}^{(k-1)} + 0.5 * \Delta_{\%cost},$$

   where,
   $k =$ iteration number, and
   $\Delta_{\%cost}$ is same as above (equation 3).

   Calculate $\Delta_{cost}, \Delta_{\%cost}$ at the end of each iteration of the while loop.
   Initialize $\Delta_{cost} = 0$, $\mathbf{w} = \mathbf{0}, b = 0$ and compute $f_0(\mathbf{w}, b)$ with these values.
   For this method, use $\eta = 0.0001, \epsilon = 0.001$.

3. **Mini batch gradient descent**: Go through the dataset in batches of predetermined size and update the parameters as follows:

   Randomly shuffle the training data
   $l = 0, k = 0$
   while convergence criteria not reached **do**
     for $j = 1, ..., d$ **do**
       Update $w^{(j)} \leftarrow w^{(j)} - \eta \nabla_{w^{(j)}} f_l(\mathbf{w}, b)$
     end for
     Update $b \leftarrow b - \eta \nabla_b f_l(\mathbf{w}, b)$
     Update $l \leftarrow (l + 1) \mod ((n + batch\_size - 1)/batch\_size)$
     Update $k \leftarrow k + 1$
   end while

   where,
   n is the number of samples in the training data,
   d is the dimensions of $\mathbf{w}$,
   η is the learning rate,
   $batch\_size$ is the number of training samples considered in each batch, and
   $\nabla_{w^{(j)}} f_l(\mathbf{w}, b)$ is defined for a batch of training samples as follows:

$$\nabla_{w^{(j)}} f_l(\mathbf{w}, b) = \frac{\partial f_l(\mathbf{w}, b)}{\partial w^{(j)}} = w^{(j)} + C \sum_{i=l*batch\_size+1}^{min(n,(l+1)*batch\_size)} \frac{\partial L(x_i, y_i)}{\partial w^{(j)}},$$

   The convergence criteria is $\Delta_{cost}^{(k)} < \epsilon$, where

$$\Delta_{cost}^{(k)} = 0.5 * \Delta_{cost}^{(k-1)} + 0.5 * \Delta_{\%cost},$$

   k = iteration number,
   and $\Delta_{\%cost}$ is same as above (equation 3).

   Calculate $\Delta_{cost}, \Delta_{\%cost}$ at the end of each iteration of the while loop.
   Initialize $\Delta_{cost} = 0$, $\mathbf{w} = \mathbf{0}, b = 0$ and compute $f_0(\mathbf{w}, b)$ with these values.
   For this method, use $\eta = 0.00001, \epsilon = 0.01$, batch_size $= 20$.

(a) [5 Points]

Notice that we have not given you the equation for, $\nabla_b f(\mathbf{w}, b)$.

Task: What is $\nabla_b f(\mathbf{w}, b)$ used for the Batch Gradient Descent Algorithm?

(Hint: It should be very similar to $\nabla_{w^{(j)}} f(\mathbf{w}, b)$.)

★ **SOLUTION:**

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

where

$$\frac{\partial L(x_i, y_i)}{\partial b} = \begin{cases} 0 & \text{if } y_i \left(\mathbf{x}_1 \cdot \mathbf{w} + b\right) \geq 1 \\ -y_i & \text{otherwise.} \end{cases}$$

(b) [25 Points]

Task: Implement the SVM algorithm for all of the above mentioned gradient descent techniques.

Use $C = 100$ for all the techniques. For all other parameters, use the values specified in the description of the technique. **Note:** update w in iteration $i + 1$ using the values computed in iteration i. Do not update using values computed in the current iteration!

Run your implementation on the data set in `q1/data`. The data set contains the following files :

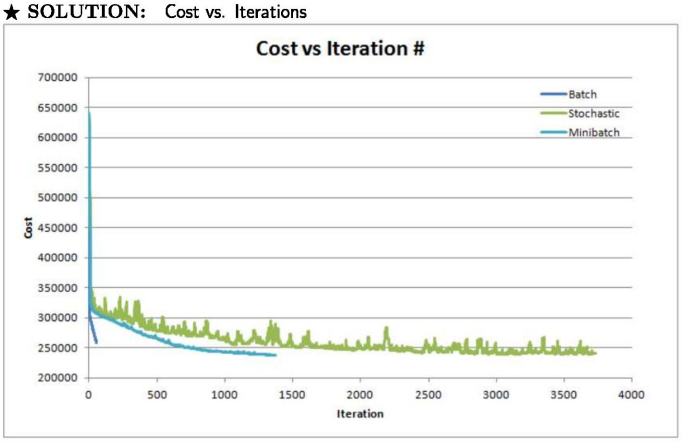
1. `features.txt` : Each line contains features (comma-separated values) for a single datapoint. It has 6414 datapoints (rows) and 122 features (columns).
2. `target.txt` : Each line contains the target variable (y = -1 or 1) for the corresponding row in `features.txt`.

Task: Plot the value of the cost function $f_k(\mathbf{w}, b)$ vs. the number of iterations (k). Report the total time taken for convergence by each of the gradient descent techniques. What do you infer from the plots and the time for convergence?

The diagram should have graphs from all the three techniques on the same plot.

As a sanity check, Batch GD should converge in 10-300 iterations and SGD between 500-3000 iterations with Mini Batch GD somewhere in-between. However, the number of iterations may vary greatly due to randomness. If your implementation consistently takes longer though, you may have a bug.

★ **SOLUTION:** Cost vs. Iterations



Sample Runtimes (lots of variation based on implementation): Batch: 430s SGD: 34s Mini Batch: 44s

The plot for BGD should take about 60 iterations to converge. It should be monotonic since it is calculating the true gradient for the whole dataset. The plot for SGD and MBGD should exhibit some randomness in converging to around 250,000. Generally, MBGD will converge in fewer iterations than SGD because it has more information in making each update step.

Depending on implementation, wall clock times may vary. If BGD took much less time than the randomized algorithms, it's likely because it was vectorized and the others were not. This is what you might see if the dataset is small enough to fit in memory. Another reasonable explanation is that we were calculating cost at every iteration so per-iteration time was dominated by the cost calculation. If BGD took more time than the others, this was likely because it saw more of the data at each step. This was common in solutions that did not vectorized.

Most reasonable solutions were accepted. No attempt at an explanation was not.

What to submit

(i) Equation for $\nabla_b f(\mathbf{w}, b)$. [part (a)]

(ii) Plot of $f_k(\mathbf{w}, b)$ vs. the number of updates (k). Total time taken for convergence by each of the gradient descent techniques. Interpretation of plot and convergence times. [part (b)]

(iii) Submit the code on snap submission website. [part (b)]

2 Decision Tree Learning (20 points)

In this problem, we want to construct a decision tree to find out if a person will enjoy beer.

Definitions. Let there be k binary-valued attributes in the data.

We pick an attribute that maximizes the gain at each node:

$$G = I(D) - (I(D_L) + I(D_R)); \tag{4}$$

where D is the given dataset, and D_L and D_R are the sets on left and right hand-side branches after division. Ties may be broken arbitrarily.

There are three commonly used impurity measures used in binary decision trees: Entropy, Gini index, and Classification Error. In this problem, we use Gini index and define $I(D)$ as follows[1]:

$$I(D) = |D| \times \left(1 - \sum_i p_i^2\right),$$

where:

- $|D|$ is the number of items in D;
- $1 - \sum_i p_i^2$ is the gini index;
- p_i is the probability distribution of the items in D, or in other words, p_i is the fraction of items that take value $i \in \{+, -\}$. Put differently, p_+ is the fraction of positive items and p_- is the fraction of negative items in D.

Note that this intuitively has the feel that the more evenly-distributed the numbers are, the lower the $\sum_i p_i^2$, and the larger the impurity.

(a) [10 Points]

Let $k = 3$. We have three binary attributes that we could use: "likes wine", "likes running" and "likes pizza". Suppose the following:

- There are 100 people in sample set, 40 of whom like beer and 60 who don't.
- Out of the 100 people, 50 like wine; out of those 50 people who like wine, 20 like beer.
- Out of the 100 people, 30 like running; out of those 30 people who like running, 20 like beer.

[1] As an example, if D has 10 items, with 4 positive items (*i.e.* 4 people who enjoy beer), and 6 negative items (*i.e.* 6 who do not), we have $I(D) = 10 \times (1 - (0.16 + 0.36))$.

- Out of the 100 people, 80 like pizza; out of those 80 people who like pizza, 30 like beer.

Task: What are the values of G (defined in Equation 4) for wine, running and pizza attributes? Which attribute would you use to split the data at the root if you were to maximize the gain G using the gini index metric defined above?

★ **SOLUTION:** The values of G for wine, running and pizza are 0, 6.1, 0.5 respectively. So Running attribute should be used for the decision at the root.

Before we do any splitting, the impurity is $I(D) = 100 \cdot (1 - (0.4^2 + 0.6^2)) = 48$.

Wine: Suppose we used the wine attribute. 20 out of 50 wine-drinkers like beer, and 20 out of 50 non-wine-drinkers like beer. So the impurity on the "likes wine" side of the decision tree is $I(D_L) = 50 \cdot (1 - (0.4^2 + 0.6^2)) = 24$, and the impurity on the "doesn't like wine" side of the decision tree is $I(D_R) = 50 \cdot (1 - (0.4^2 + 0.6^2)) = 24$. So there would be no reduction in impurity, which makes sense because the fraction of beer drinkers is exactly the same in both groups.

Running: Suppose we used the running attribute. 20 out of 30 runners like beer, and 20 out of 70 non-runners like beer. So the impurity on the "likes running" side of the decision tree is $30 \cdot (1 - (0.66666^2 + 0.33333^2)) = 13.3333$, and the impurity on the other side is $70 \cdot (1 - (0.28571^2 + 0.71429^2)) = 28.5712$. So the total impurity decreases by $48 - 13.3333 - 28.5712 = 6.0955$.

Pizza: Suppose we used the pizza attribute. 30 out of 80 pizza-lovers like beer, and 10 out of 20 non-pizza-lovers like beer. So the impurity on the "likes pizza" side of the decision tree is $80 \cdot (1 - (0.375^2 + 0.625^2)) = 37.5$, and the impurity on the "doesn't like pizza" side is $20 \cdot (1 - (0.5^2 + 0.5^2)) = 10$. So the total impurity decreases by $48 - 37.5 - 10 = 0.5$.

Therefore, we should use the **Running attribute**.

(b) [10 Points]

Let's consider the following example:

- There are 100 attributes with binary values $a_1, a_2, a_3, \ldots, a_{100}$.
- Let there be one example corresponding to each possible assignment of 0's and 1's to the values $a_1, a_2, a_3 \ldots, a_{100}$. (Note that this gives us 2^{100} training examples.)
- Let the values taken by the target variable y depend on the values of a_1 for 99% of the datapoints. More specifically, of all the datapoints where $a_1 = 1$, let 99% of them are labeled $+$. Similarly, of all the datapoints where $a_1 = 0$, let 99% of them are labeled with $-$. (Assume that the values taken by y depend on $a_2, a_3, \ldots, a_{100}$ for fewer than 99% of the datapoints.)

- Assume that we build a complete binary decision tree (*i.e.*, we use values of all attributes).

Task: Explain what the decision tree will look like. (A one line explanation will suffice.) Also, in 2-3 sentences, identify what the desired decision tree for this situation should look like to avoid overfitting, and why.(The desired decision tree isn't necessarily a complete binary decision tree)

★ **SOLUTION:** a_1 will be at the root. For the left branch ($a_1 = 0$), around 99% of the leaves will be negative, while for the right branch, around 99% of the leaves will be positive. It is hard to say which of the rest of the attributes will be at each node in the tree, except that each path would consider all the attributes (Assumption 4 in the question above). Also the desired decision tree which avoids overfitting would have a single decision at the root corresponding to a_1 (since none of the other attributes are predictive of the outcome, and the 1% is likely to be noise).

What to submit

(i) Values of G for wine, running and pizza attributes. [part (a)]

(ii) The attribute you would use for splitting the data at the root. [part (a)]

(iii) Explain what the decision tree looks like in the described setting. Explain how a decision tree should look like to avoid overfitting. (1-2 lines each) [part (b)]

3 Clustering Data Streams (20 points)

Introduction. In this problem, we study an approach for clustering massive data streams. We will study a framework for turning an approximate clustering algorithm into one that can work on data streams, *i.e.*, one which needs a small amount of memory and a small number of (actually, just one) passes over the data. As the instance of the clustering problem, we will focus on the k-means problem.

Definitions. Before going into further details, we need some definitions:

- The function $d : \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}^+$ denotes the Euclidean distance:
$$d(x, y) = ||x - y||_2.$$
- For any $x \in \mathbb{R}^p$ and $T \subset \mathbb{R}^p$, we define:
$$d(x, T) = \min_{z \in T} \{d(x, z)\}.$$

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

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

- Finally, if for all $x \in S$ we have $w(x) = 1$, we simply denote $\text{cost}_w(S, T)$ by $\text{cost}(S, T)$.

Reminder: k-means clustering. The k-means clustering problem is as follows: given a subset $S \subset \mathbb{R}^p$, and an integer k, find the set T (with $|T| = k$), which minimizes $\text{cost}(S, T)$. If a weight function w is also given, the k-means objective would be to minimize $\text{cost}_w(S, T)$, and we call the problem the weighted k-means problem.

Strategy for clustering data streams. We assume we have an algorithm ALG which is an α-approximate weighted k-means clustering algorithm (for some $\alpha > 1$). In other words, given any $S \subset \mathbb{R}^p$, $k \in \mathbb{N}$, and a weight function w, ALG returns a set $T \subset \mathbb{R}^p$, $|T| = k$, such that:

$$\text{cost}_w(S, T) \leq \alpha \min_{|T'|=k} \{\text{cost}_w(S, T')\}.$$

We will see how we can use ALG as a building block to make an algorithm for the k-means problem on data streams.

The basic idea here is that of divide and conquer: if S is a huge set that does not fit into main memory, we can read a portion of it that does fit into memory, solve the problem on this subset (*i.e.*, do a clustering on this subset), record the result (*i.e.*, the cluster centers and some corresponding weights, as we will see), and then read a next portion of S which is again small enough to fit into memory, solve the problem on this part, record the result, etc. At the end, we will have to combine the results of the partial problems to construct a solution for the main big problem (*i.e.*, clustering S).

To formalize this idea, we consider the following algorithm, which we denote as ALGSTR:

- Partition S into ℓ parts $S_1, \ldots, S_\ell$.
- For each $i = 1$ to ℓ, run ALG on S_i to get a set of k centers $T_i = \{t_{i1}, t_{i2}, \ldots, t_{ik}\}$, and assume $\{S_{i1}, S_{i2}, \ldots, S_{ik}\}$ is the corresponding clustering of S_i (*i.e.*, $S_{ij} = \{x \in S_i |\, d(x, t_{ij}) < d(x, t_{ij'}) \; \forall j' \neq j, 1 \leq j' \leq k\}$).
- Let $\widehat{S} = \bigcup_{i=1}^{\ell} T_i$, and define weights $w(t_{ij}) = |S_{ij}|$.
- Run ALG on $\widehat{S}$ with weights w, to get k centers T.
- Return T.

Now, we analyze this algorithm. Assuming $T^* = \{t_1^*, \ldots, t_k^*\}$ to be the optimal k-means solution for S (that is, $T^* = \text{argmin}_{|T'|=k}\{\text{cost}(S, T')\}$), we would like to compare $\text{cost}(S, T)$ (where T is returned by ALGSTR) with $\text{cost}(S, T^*)$.

A small fact might be useful in the analysis below: for any $(a, b) \in \mathbb{R}^+$ we have:

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

(a) [5pts]

First, we show that the cost of the final clustering can be bounded in terms of the total cost of the intermediate clusterings:

Task: Prove that:

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

Hint: You might want to use Triangle Inequality for Euclidean distance d.

★ **SOLUTION:** Let $T(x) = \text{argmin}_{t \in T} d(t, x)$. By triangle inequality, for any $x \in S_{ij}$ $(1 \leq i \leq \ell, 1 \leq j \leq k)$: $d(x, T) = d(x, T(x)) \leq d(x, T(t_{ij})) \leq d(x, t_{ij}) + d(t_{ij}, T(t_{ij})) = d(x, t_{ij}) + d(t_{ij}, T)$, and hence by the small fact given in the introduction, $d(x, T)^2 \leq 2d(x, t_{ij})^2 + 2d(t_{ij}, T)^2$. Summing up over all i, j, x gives the result.

(b) [5pts]

So, to bound the cost of the final clustering, we can bound the terms on the right hand side of the inequality in part (a). Intuitively speaking, we expect the second term to be small compared to $\text{cost}(S, T^*)$, because T^* only uses k centers to represent the data set (S), while the T_i's, in total, use $k\ell$ centers to represent the same data set (and $k\ell$ is potentially much bigger than k). We show this formally:

Task: Prove that:

$$\sum_{i=1}^{\ell} \text{cost}(S_i, T_i) \leq \alpha \cdot \text{cost}(S, T^*).$$

★ **SOLUTION:** Assume T_i^* is the optimal clustering for S_i $(1 \leq i \leq \ell)$. Then, $\text{cost}(S_i, T_i) \leq \alpha \cdot \text{cost}(S_i, T_i^*) \leq \alpha \cdot \text{cost}(S_i, T^*)$. Summing up over all i gives the result.

(c) [10pt]

Prove that ALGSTR is a $(4\alpha^2 + 6\alpha)$-approximation algorithm for the k-means problem.

Task: Prove that:

$$\text{cost}(S, T) \leq (4\alpha^2 + 6\alpha) \cdot \text{cost}(S, T^*).$$

Hint: You might want to first prove two useful facts, which help bound the first term on the right hand side of the inequality in part (a):

$$\text{cost}_w(\widehat{S}, T) \leq \alpha \cdot \text{cost}_w(\widehat{S}, T^*).$$

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

★ **SOLUTION:** If $\widehat{T}^*$ is the best clustering for $\widehat{S}$, then: $\text{cost}_w(\widehat{S}, T) \leq \alpha \cdot \text{cost}_w(\widehat{S}, \widehat{T}^*) \leq \alpha \cdot \text{cost}_w(\widehat{S}, T^*)$

Similarly to part (a), for any $x \in S_{ij}$, $(1 \leq i \leq \ell, 1 \leq j \leq k)$, we have: $d(t_{ij}, T^*)^2 \leq 2d(t_{ij}, x)^2 + 2d(x, T^*)^2$. Summing up over all i, j, x gives the result for ii.

To prove the last fact, you just have to use the previous parts.

Additional notes: We have shown above that ALGSTR is a $(4\alpha^2 + 6\alpha)$-approximation algorithm for the k-means problem. Clearly, $4\alpha^2 + 6\alpha > \alpha$, so ALGSTR has a somewhat worse approximation guarantee than ALG (with which we started). However, ALGSTR is better suited for the streaming application, as not only it takes just one pass over the data, but also it needs a much smaller amount of memory.

Assuming that ALG needs $\Theta(n)$ memory to work on an input set S of size n (note that just representing S in memory will need $\Omega(n)$ space), if we partitioning S into $\sqrt{n/k}$ equal parts, ALGSTR can work with only $O(\sqrt{nk})$ memory. (Like in the rest of the problem, k represents the number of clusters per partition.)

Note that for typical values of n and k, assuming $k \ll n$, we have $\sqrt{nk} \ll n$. For instance, with $n = 10^6$, and $k = 100$, we have $\sqrt{nk} = 10^4$, which is 100 times smaller than n.

What to submit

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

(b) Proof that $\sum_{i=1}^{\ell} \text{cost}(S_i, T_i) \leq \alpha \cdot \text{cost}(S, T^*)$.

(c) Proof that $\text{cost}(S, T) \leq (4\alpha^2 + 6\alpha) \cdot \text{cost}(S, T^*)$.

4 Data Streams (30 points)

In this problem, we study an approach to approximating the frequency of occurrences of different items in a data stream. Assume $S = \langle a_1, a_2, \ldots, a_t \rangle$ is a data stream of items from

the set $\{1, 2, \ldots, n\}$. Assume for any $1 \leq i \leq n$, $F[i]$ is the number of times i has appeared in S. We would like to have good approximations of the values $F[i]$ $(1 \leq i \leq n)$ at all times.

A simple way to do this is to just keep the counts for each item $1 \leq i \leq n$ separately. However, this will require $\mathcal{O}(n)$ space, and in many applications (e.g., think online advertising and counts of user's clicks on ads) this can be prohibitively large. We see in this problem that it is possible to approximate these counts using a much smaller amount of space. To do so, we consider the algorithm explained below.

Strategy. The algorithm has two parameters $\delta, \epsilon > 0$. It picks $\lceil \log \frac{1}{\delta} \rceil$ independent hash functions:

$$\forall j \in \left[\!\left[1; \left\lceil \log \frac{1}{\delta} \right\rceil\right]\!\right], \quad h_j : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, \left\lceil \frac{e}{\epsilon} \right\rceil\},$$

where log denotes natural logarithm. Also, it associates a count $c_{j,x}$ to any $1 \leq j \leq \lceil \log \frac{1}{\delta} \rceil$ and $1 \leq x \leq \lceil \frac{e}{\epsilon} \rceil$. In the beginning of the stream, all these counts are initialized to 0. Then, upon arrival of each a_k $(1 \leq k \leq t)$, each of the counts $c_{j,h_j(a_k)}$ $(1 \leq j \leq \lceil \log \frac{1}{\delta} \rceil)$ is incremented by 1.

For any $1 \leq i \leq n$, we define $\tilde{F}[i] = \min_j \{c_{j,h_j(i)}\}$. We will show that $\tilde{F}[i]$ provides a good approximation to $F[i]$.

Memory cost. Note that this algorithm only uses $\mathcal{O}\left(\frac{1}{\epsilon} \log \frac{1}{\delta}\right)$ space.

Properties. A few important properties of the algorithm presented above:

- For any $1 \leq i \leq n$:
$$\tilde{F}[i] \geq F[i].$$
- For any $1 \leq i \leq n$ and $1 \leq j \leq \lceil \log(\frac{1}{\delta}) \rceil$:
$$\mathsf{E}\left[c_{j,h_j(i)}\right] \leq F[i] + \frac{\epsilon}{e}(t - F[i]).$$

(a) [10 Points]

Prove that:

$$\mathsf{Pr}\left[\tilde{F}[i] \leq F[i] + \epsilon t\right] \geq 1 - \delta.$$

Hint: Use Markov inequality and the property of independence of hash functions.

★ **SOLUTION:**

$$\begin{aligned} Pr(\tilde{F}[i] \leq F[i] + \epsilon t) &= 1 - Pr(\tilde{F}[i] > F[i] + \epsilon t) \\ &= 1 - Pr(c_{j,h_j(i)} > F[i] + \epsilon t, \forall 1 \leq j \leq \lceil \log(\frac{1}{\delta}) \rceil) \\ &= 1 - \prod_{j=1}^{\lceil \log(\frac{1}{\delta}) \rceil} Pr(c_{j,h_j(i)} > F[i] + \epsilon t) \end{aligned}$$

where the last equality is using the independence of different hash functions. The result then follows by noticing that for any j, by Markov's inequality and the properties mentioned above:

$$Pr(c_{j,h_j(i)} > F[i] + \epsilon t) \leq Pr(c_{j,h_j(i)} \geq F[i] + \epsilon t) = Pr(c_{j,h_j(i)} - F[i] \geq \epsilon t) \leq \frac{E[c_{j,h_j(i)} - F[i]]}{\epsilon t} \leq \frac{1}{e}$$

Based on the proof in part (a) and the properties presented earlier, it can be inferred that $\tilde{F}[i]$ is a good approximation of $F[i]$ for any item i such that $F[i]$ is not very small (compared to t). In many applications (*e.g.*, when the values $F[i]$ have a heavy-tail distribution), we are indeed only interested in approximating the frequencies for items which are not too infrequent. We next consider one such application.

(b) [20 Points]

Warning. This implementation question requires substantial computation time Python implementation reported to take 15min - 1 hour. Therefore, we advise you to start early.

Dataset. The dataset in **q4/data** contains the following files:

1. `words_stream.txt` Each line of this file is a number, corresponding to the ID of a word in the stream.
2. `counts.txt` Each line is a pair of numbers separated by a tab. The first number is an ID of a word and the second number is its associated exact frequency count in the stream.
3. `words_stream_tiny.txt` and `counts_tiny.txt` are smaller versions of the dataset above that you can use for debugging your implementation.
4. `hash_params.txt` Each line is a pair of numbers separated by a tab, corresponding to parameters a and b which you may use to define your own hash functions (See explanation below).

Instructions. Implement the algorithm and run it on the dataset with parameters $\delta = e^{-5}, \epsilon = e \times 10^{-4}$. (Note: with this choice of δ you will be using 5 hash functions - the 5 pairs (a, b) that you'll need for the hash functions are in `hash_params.txt`). Then for each distinct word i in the dataset, compute the relative error $E_r[i] = \frac{\tilde{F}[i]-F[i]}{F[i]}$ and plot these values as a function of the exact word frequency $\frac{F[i]}{t}$. (**You do not have to implement the algorithm in Spark.**)

The plot should use a logarithm scale both for the x and the y axes, and there should be ticks to allow reading the powers of 10 (e.g. 10^{-1}, 10^0, 10^1 etc...). The plot should have a title, as well as the x and y axes. The exact frequencies $F[i]$ should be read from the counts file. Note that words of low frequency can have a very large relative error. That is not a bug in your implementation, but just a consequence of the bound we proved in question (a).

Answer the following question by reading values from your plot: What is an approximate condition on a word frequency in the document to have a relative error below $1 = 10^0$?

Hash functions. You may use the following hash function (see example pseudo-code), with $p = 123457$, a and b values provided in the hash params file and `n_buckets` (which is equivalent to $\lceil \frac{e}{\epsilon} \rceil$) chosen according to the specification of the algorithm. In the provided file, each line gives you a, b values to create one hash function.

```
# Returns hash(x) for hash function given by parameters a, b, p and n_buckets
def hash_fun(a, b, p, n_buckets, x)
{
y = x [modulo] p
hash_val = (a*y + b) [modulo] p
return hash_val [modulo] n_buckets
}
```

Note: This hash function implementation produces outputs of value from 0 to (`n_buckets` − 1), which is different from our specification in the **Strategy** part. You can either keep the range as $\{0, ..., \texttt{n\_buckets} - 1\}$, or add 1 to the hash result so the value range becomes $\{1, ..., \texttt{n\_buckets}\}$, as long as you stay consistent within your implementation.

What to submit

(i) Proof that $\mathsf{Pr}\left[\tilde{F}[i] \leq F[i] + \epsilon t\right] \geq 1 - \delta$. [part (a)]

(ii) Log-log plot of the relative error as a function of the frequency. Answer for which word frequencies is the relative error below 1. [part (b)]

(iii) Submit the code on snap submission site. [part (b)]

★ **SOLUTION:** The algorithm works best for frequent words (see figure below). Words with frequency over 10^{-5} tend to have a relative error below 1 (which means that the estimate count is less than the double of the actual count). This coincides with what was derived in part (c).

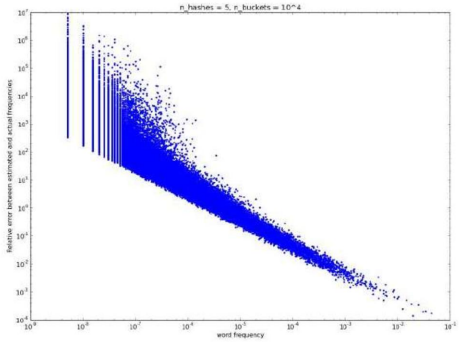


Figure 1: Results for the DS algorithm