CSE 547: Machine Learning for Big Data **Spring 2019**

Problem Set 4

Please read the homework submission policies here.

Assignment Submission All students should submit their assignments electronically via GradeScope. Students may typeset or scan their **neatly written** homeworks (points **will** be deducted for illegible submissions). Simply sign up on Gradescope and use the course code 97EWEW. Please use your UW NetID if possible.

For the non-coding component of the homework, you should upload a PDF rather than submitting as images. We will use Gradescope for the submission of code as well. Please make sure to tag each part correctly on Gradescope so it is easier for us to grade. There will be a small point deduction for each mistagged page and for each question that includes code. Put all the code for a single question into a single file and upload it. Only files in text format (e.g. .txt, .py, .java) will be accepted. **There will be no credit for coding questions without submitted code on Gradescope, or for submitting it after the deadline**, so please remember to submit your code.

Late Day Policy All students will be given two no-questions-asked late periods, but only one late period can be used per homework and cannot be used for project deliverables. A late-period lasts 48 hours from the original deadline (so if an assignment is due on Thursday at 11:59 pm, the late period goes to the Saturday at 11:59pm Pacific Time).

Academic Integrity We take academic integrity extremely seriously. We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions and the code independently. In addition, each student should write down the set of people whom they interacted with.

Discussion Group (People with whom you discussed ideas used in your answers):

On-line or hardcopy documents used as part of your answers:

I acknowledge and accept the Academic Integrity clause.

*(Signed)*_____

1 Implementation of SVM via Gradient Descent (25 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, given a dataset of n samples $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where every d-dimensional feature vector $\mathbf{x}_i \in \mathbb{R}^d$ is associated with a label $y_i \in \{-1, 1\}$, to estimate the $\mathbf{w}$, b of the soft margin SVM, we can minimize the cost function:

$$f(\mathbf{w}, b) = \frac{1}{2}\|\mathbf{w}\|_2^2 + C\sum_{i=1}^{n} \max\left\{0, 1 - y_i\left(\mathbf{w}\cdot\mathbf{x}_i + b\right)\right\} \tag{1}$$

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.

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

and

$$\nabla_b f(\mathbf{w}, b) = \frac{\partial f(\mathbf{w}, b)}{\partial b} = C\sum_{i=1}^{n} \frac{\partial L(x_i, y_i)}{\partial b} \tag{3}$$

where

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

and

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

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:

Algorithm 1 Batch Gradient Descent (BGD)

Parameters: learning rate η.

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

Note that in Algorithm 1, the values of $\nabla_{w^{(j)}} f(\mathbf{w}, b)$ and $\nabla_b f(\mathbf{w}, b)$ are computed by equations 2 and 3 repectively.

The *convergence criterion* for algorithm 1 is $\Delta_{\%cost} < \varepsilon$, where

$$\Delta_{\%cost} = \frac{|f_{k-1}(\mathbf{w}, b) - f_k(\mathbf{w}, b)| \times 100}{f_{k-1}(\mathbf{w}, b)}. \tag{4}$$

where $f_k(\mathbf{w}, b)$ is the regularized loss (equation 1) at k-th iteration.

For this method, set $\eta = 3 \cdot 10^{-7}$, $\varepsilon = 0.25$. Initialize $\mathbf{w} = \mathbf{0}$, $b = 0$ and compute $f_0(\mathbf{w}, b)$ with these values. Compute $\Delta_{\%cost}$ at the end of each iteration of the while loop.

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

Algorithm 2 Stochastic Gradient Descent (SGD)

Parameters: learning rate η.

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

In Algorithm 2, $\nabla_{w^{(j)}} f_i(\mathbf{w}, b)$ and $\nabla_b f_i(\mathbf{w}, b)$ are defined for the i-th training sample as follows:

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

$$\nabla_b f_i(\mathbf{w}, b) = \frac{\partial f(\mathbf{w}, b)}{\partial b} = C \frac{\partial L(x_i, y_i)}{\partial b}$$

The *convergence criterion* here is $\Delta_{cost}^{(k)} < \varepsilon$, where

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

where k is the iteration number and $\Delta_{\%cost}$ is same as above (equation 4).

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

3. **Mini-Batch Gradient Descent**: Go through the dataset in batches of predetermined size and update the parameters as follows:

Algorithm 3 Mini-Batch Gradient Descent (MBGD)

Parameters: learning rate η, batch size B.

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 $l \leftarrow (l + 1 \mod \lceil n/B \rceil)$
  Update $k \leftarrow k + 1$
end while

In Algorithm 3, $\nabla_{w^{(j)}} f_l(\mathbf{w}, b)$ and $\nabla_b f_l(\mathbf{w}, b)$ are defined for the l-th mini-batch as follows:

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

$$\nabla_b f_l(\mathbf{w}, b) = \frac{\partial f_l(\mathbf{w}, b)}{\partial b} = C \sum_{i=l \cdot B+1}^{min\{n,(l+1)\cdot B\})} \frac{\partial L(x_i, y_i)}{\partial b}$$

The *convergence criterion* here is $\Delta_{cost}^{(k)} < \varepsilon$, where

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

where k is the iteration number and $\Delta_{\%cost}$ is same as above (equation 4).

For this method, use $\eta = 10^{-5}$, $\varepsilon = 0.01$ and $B = 20$. Initialize $\Delta_{cost}^{(0)} = 0$, $\mathbf{w} = \mathbf{0}$, $b = 0$ and compute $f_0(\mathbf{w}, b)$ with these values. Calculate $\Delta_{cost}^{(k)}$ and $\Delta_{\%cost}$ at the end of each iteration of the while loop.

(a) [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 the features (comma-separated values) of a single sample. It has 6414 samples (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 iteration number 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, it may have a bug.

What to submit

(i) 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 (a)]

(ii) Submit the code to Gradescope. [part (a)]

2 Decision Tree Learning (15 points)

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

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

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

$$G = I(D) - (I(D_L) + I(D_R)) \tag{5}$$

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 impurity and define $I(D)$ as follows[1]:

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

where:

- $|D|$ is the number of items in D.
- $1 - \sum_i p_i^2$ is the Gini impurity.
- p_i is the probability that an element sampled uniformly at random from D will have label $i \in \{+, -\}$. Explicitly, 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 tea", "likes hiking" and "likes chocolate ice cream". Suppose the following:

- There are 100 people in our sample set, 75 of whom like coffee and 25 who don't.
- Out of the 100 people, 50 like tea; out of those 50 people who like tea, 40 like coffee.
- Out of the 100 people, 70 like hiking; out of those 70 people who like hiking, 60 like coffee.
- Out of the 100 people, 80 like chocolate ice cream; out of those 80 people who like chocolate ice cream, 60 like coffee.

Task: What are the values of G (defined in Equation 5) for chocolate ice cream, tea and hiking attributes? Which attribute would you use to split the data at the root if you were to maximize the gain G using the Gini impurity metric defined above?

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

(b) [5 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 samples. More specifically, of all the samples where $a_1 = 1$, let 99% of them are labeled $+$. Similarly, of all the samples 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 samples).
- 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)

What to submit

(i) Values of G for chocolate ice cream, tea and hiking 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 Data Streams I (35 points)

You are an astronomer at the Space Telescope Science Institute in Baltimore, Maryland, in charge of the *petabytes* of imaging data they recently obtained. According to the news report linked in the previous sentence, "*...The amount of imaging data is equivalent to two billion selfies, or 30,000 times the total text content of Wikipedia. The catalog data is 15 times the volume of the Library of Congress.*"

This data stream has images of everything out there in the universe, ranging from stars, galaxies, asteroids, to all kinds of awesome exploding/moving objects. Your task is to determine the approximate frequencies of occurrences of different (unique) items in this data stream.

We now introduce our notation for this problem. Let $S = \langle a_1, a_2, \ldots, a_t \rangle$ be the given data stream of length t. Let us denote the items in this data stream as being from the set $\{1, 2, \ldots, n\}$. For any $1 \leq i \leq n$, we denote $F[i]$ to be the number of times i has appeared in S. Our goal is then to have good approximations of the values $F[i]$ for all $1 \leq i \leq n$ at all times.

The naïve 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 which, in our application, is clearly infeasible. We shall see that it is possible to approximate these counts using a much smaller amount of space. To do so, we consider the algorithm explained below.

Algorithm. The algorithm has two parameters δ and $\varepsilon > 0$, and $\left\lceil \log \frac{1}{\delta} \right\rceil$ *independent* hash functions

$$h_j : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, \left\lceil \frac{e}{\varepsilon} \right\rceil\}.$$

Note that in this problem, log denotes the natural logarithm. For each bucket b of each hash function j, the algorithm has a counter $c_{j,b}$ that is initialized to zero.

As each element i arrives in the data stream, it is hashed by the j hash functions, and the count $c_{j,h_j(i)}$ is incremented by 1.

For any $1 \leq i \leq n$, we define $\widetilde{F}[i] = \min_j \{c_{j,h_j(i)}\}$ as our estimate of $F[i]$.

Task. The goal is to show that $\widetilde{F}[i]$ as defined above provides a good estimate of $F[i]$.

(a) [2 Points]

What is the memory usage of this algorithm (in Big-$\mathcal{O}$ notation)? Give a one or two line justification for the value you provide.

(b) [3 Points]

Justify that for any $1 \leq i \leq n$:

$$\widetilde{F}[i] \geq F[i].$$

(c) [10 Points]

Prove that for any $1 \leq i \leq n$ and $1 \leq j \leq \lceil \log(\frac{1}{\delta}) \rceil$:

$$\mathsf{E}\left[c_{j,h_j(i)}\right] \leq F[i] + \frac{\varepsilon}{e} t.$$

(d) [10 Points]

Prove that:

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

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

Based on the proofs in parts (b) and (d), it can be inferred that $\widetilde{F}[i]$ 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.

(e) [10 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 $\delta = e^{-5}, \varepsilon = e \times 10^{-4}$. (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 $E_r[i] = \frac{\widetilde{F}[i]-F[i]}{F[i]}$ 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 $\lceil \frac{e}{\varepsilon} \rceil$) 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) Expression for the memory usage of the algorithm and justification. [part (a)]

(ii) Proofs for parts (b)-(d).

(iii) 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 (e)]

(iv) Submit the code to Gradescope. [part (e)]

4 Data Streams II (25 points)

We are in the same setup as the previous part, working with the stream $S = \langle a_1, a_2, \ldots, a_t \rangle$ consisting of items from the set $\{1, 2, \ldots, n\}$; the frequency of element i is again denoted by $F[i]$. Impressed by the quality of your estimator for frequency of occurences of different objects in the telescope's data stream, you have been now tasked with estimating more sophisticated summary statistics from this stream. We'll now explore how to estimate the

sum of *squared* frequencies of all the items in the data stream. That is, we wish to estimate $M = \sum_{i=1}^{n} (F[i])^2$. Here's a proposed algorithm.

Algorithm.

- Fix a function $h : [n] \to \{\pm 1\}$ that associates each item in the data stream with a random sign.
- Initialize $Z = 0$.
- Every time an element j appears in the data stream, add $h(j)$ to Z. That is, increment Z if $h(j) = +1$ and decrement Z if $h(j) = -1$.
- After processing all the t elements in the data stream, return the estimate $X = Z^2$

Note that since we fix h *before* we receive the data stream, an element j is treated consistently every time it shows up: Z is either incremented every time j shows up or is decremented every time j shows up. In the end, element j contributes $h(j) \cdot F[j]$ to the final value of Z.

(a) [10 Points]

Prove that

$$\mathbb{E}_h[X] = M$$

(b) [15 Points]

Prove that

$$\text{Var}(X) \leq 4M^2$$

Thus, part (a) shows that the estimator designed in the given algorithm is *unbiased*, while part (b) gives a bound on its variance.

What to submit

(i) Proof that $\mathbb{E}_h[X] = M$. [parts (a)]

(ii) Proof that $\text{Var}(X) \leq 4M^2$. [part (b)]