CSE 547: Machine Learning for Big Data **Spring 2024**

Problem Set 4

Please read the homework submission policies here.

Assignment Submission All students should submit their assignments electronically via GradeScope. No handwritten work will be accepted. Math formulas **must** be typeset using LaTeX or other word processing software that supports mathematical symbols (E.g. Google Docs, Microsoft Word). Simply sign up on Gradescope and use the course code V84RPB. 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.

Coding You may use any programming languages and standard libraries, such as NumPy and PySpark, but you may not use specialized packages and, in particular, machine learning libraries (e.g. sklearn, TensorFlow), unless stated otherwise. Ask on the discussion board whether specific libraries are allowed if you are unsure.

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 (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. Our goal for this problem is to investigate the convergence of different gradient descent methods on a sample dataset and think about the characteristics of these different methods that lead to different performances.

To recap, given a dataset of n samples $\mathcal{D} = \left\{\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)\right\}_{i=1}^{n}$, where every d-dimensional feature vector $\boldsymbol{x}^{(i)} \in \mathbb{R}^d$ is associated with a label $y^{(i)} \in \{-1, 1\}$, to estimate the parameters $\boldsymbol{\theta} = (\boldsymbol{w}, b)$ of the soft margin SVM, we can minimize the loss function:

$$\begin{aligned} f(\boldsymbol{w}, b; \mathcal{D}) &= \frac{1}{2}\|\boldsymbol{w}\|_2^2 + C \sum_{(\boldsymbol{x}^{(i)}, y^{(i)}) \in \mathcal{D}} \max\left\{0, 1 - y^{(i)}(\boldsymbol{w} \cdot \boldsymbol{x}^{(i)} + b)\right\} \\ &= \frac{1}{2}\|\boldsymbol{w}\|_2^2 + C \sum_{(\boldsymbol{x}^{(i)}, y^{(i)}) \in \mathcal{D}} L(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}) \end{aligned}$$

In order to minimize the function, we first obtain the gradient with respect to $\boldsymbol{\theta}$. The partial derivative with respect to w_j, the j-th entry in the vector $\boldsymbol{w}$, is:

$$\partial_{w_j} f(\boldsymbol{w}, b; \mathcal{D}) = \frac{\partial f(\boldsymbol{w}, b; \mathcal{D})}{\partial w_j} = w_j + C \sum_{(\boldsymbol{x}^{(i)}, y^{(i)}) \in \mathcal{D}} \frac{\partial L(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta})}{\partial w_j} \tag{1}$$

where

$$\frac{\partial L(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta})}{\partial w_j} = \begin{cases} 0 & \text{if } y^{(i)}\left(\boldsymbol{w} \cdot \boldsymbol{x}^{(i)} + b\right) \geq 1 \\ -y^{(i)} x_j^{(i)} & \text{otherwise.} \end{cases}$$

and the partial derivative with respect to b is:

$$\partial_b f(\boldsymbol{w}, b; \mathcal{D}) = \frac{\partial f(\boldsymbol{w}, b; \mathcal{D})}{\partial b} = C \sum_{(\boldsymbol{x}^{(i)}, y^{(i)}) \in \mathcal{D}} \frac{\partial L(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta})}{\partial b} \tag{2}$$

where

$$\frac{\partial L(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta})}{\partial b} = \begin{cases} 0 & \text{if } y^{(i)}\left(\boldsymbol{w} \cdot \boldsymbol{x}^{(i)} + b\right) \geq 1 \\ -y^{(i)} & \text{otherwise.} \end{cases}$$

Since the direction of the gradient is the direction of steepest ascent of the loss function, gradient descent proceeds by iteratively taking small steps along the direction opposite to the direction of gradient. The general framework of gradient descent is given in Algorithm 1.

Algorithm 1 General Gradient Descent

Parameters: learning rate η, batch size β.

Randomly shuffle the training data ▷ Only for SGD/MBGD
$k \leftarrow 0$
for $t = 1, 2, \ldots$ **do**
  $B \leftarrow \left\{\left(x^{(i)}, y^{(i)}\right) : \beta k + 1 \leq i \leq \min\{\beta(k+1), n\}\right\}$
  for $j = 1, \ldots, d$ **do**
    $w_j^{(t)} \leftarrow w_j^{(t-1)} - \eta \cdot \partial_{w_j} f(\boldsymbol{w}^{(t-1)}, b^{(t-1)}; B)$ ▷ Computed by equation 1
  end for
  $b^{(t)} \leftarrow b^{(t-1)} - \eta \cdot \partial_b f(\boldsymbol{w}^{(t-1)}, b^{(t-1)}; B)$ ▷ Computed by equation 2
  $k \leftarrow (k + 1 \mod \lceil n/\beta \rceil)$
  if convergence criteria reached **then**
    break
  end if
end for

Task: Implement the SVM algorithm using the following gradient descent variants.

For all the variants use $C = 100$, $\boldsymbol{w}^{(0)} = \mathbf{0}$, $b^{(0)} = 0$. For all other parameters, use the values specified in the description of the variant.

Note: update the parameters $\boldsymbol{w}$ and b on iteration t using the values computed on iteration $t - 1$. Do not update using values computed in the current iteration!

1. **Batch Gradient Descent (BGD)**: When the $\beta = n$, in every iteration the algorithm uses the entire dataset to compute the gradient and update the parameters.

   As a *convergence criterion* for batch gradient descent we will use $\Delta_{\%loss}^{(t)} < \varepsilon$, where

   $$\Delta_{\%loss}^{(t)} = \frac{|f(\boldsymbol{w}^{(t-1)}, b^{(t-1)}; \mathcal{D}) - f(\boldsymbol{w}^{(t)}, b^{(t)}; \mathcal{D})|}{f(\boldsymbol{w}^{(t-1)}, b^{(t-1)}; \mathcal{D})} \times 100 \tag{3}$$

   Set $\eta = 3 \cdot 10^{-7}$, $\varepsilon = 0.25$.

2. **Stochastic Gradient Descent (SGD)**: When $\beta = 1$, in every iteration the algorithm uses one training sample at a time to compute the gradient and update the parameters.

   As a *convergence criterion* for stochastic gradient descent we will use $\Delta_{loss}^{(t)} < \varepsilon$, where

   $$\Delta_{loss}^{(t)} = \tfrac{1}{2}\Delta_{loss}^{(t-1)} + \tfrac{1}{2}\Delta_{\%loss}^{(t)}, \tag{4}$$

   t is the iteration number, $\Delta_{\%loss}^{(t)}$ is same as above (equation 3) and and $\Delta_{loss}^{(0)} = 0$.

   Use $\eta = 0.0001$, $\varepsilon = 0.001$.

3. **Mini-Batch Gradient Descent (MBGD)**: In every iteration the algorithm uses mini-batches of β samples to compute the gradient and update the parameters.

As a *convergence criterion* for mini-batch gradient descent we will use $\Delta_{loss}^{(t)} < \varepsilon$, where $\Delta_{loss}^{(t)}$ is the same as above (equation 4) and $\Delta_{loss}^{(0)} = 0$

Use $\eta = 10^{-5}$, $\varepsilon = 0.01$ and $\beta = 20$.

Task: Run your implementation on the data set in `svm/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 loss function $f(\boldsymbol{w}^{(t)}, b^{(t)}; \mathcal{D})$ vs. the iteration number t starting from $t = 0$. Label the plot axes. The diagram should have graphs from all the three variants on the same plot. Report the total time (wall clock time, as opposed to the number of iterations) each of the gradient descent variants takes to converge. What do you infer from the plots and the time for convergence? Explain using 4-6 sentences.

Sanity Check 1: The value of the loss function at iteration number $t = 0$ must be around 641,400.

Sanity Check 2: 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, there might be a bug.

Sanity Check 3: The expected total run time for all 3 methods is around 5-15 minutes but might vary depending on the implementation.

What to submit

(i) Plot of $f(\boldsymbol{w}^{(t)}, b^{(t)}; \mathcal{D})$ vs. the number of updates (t) (All 3 graphs should be in the same plot). Total time taken for convergence by each of the gradient descent variants. Interpretation of plot and convergence times.

(ii) Submit the code to Gradescope.

2 node2vec on Facebook Graph Dataset (30 points)

In this question, you will implement the 2$^{\text{nd}}$-order random walk of node2vec [1], which embeds nodes with similar network neighborhoods close in the feature space. A detailed description of the problem setup and background is provided in the **template code** provided

on Colab at https://drive.google.com/file/d/1d16fTet-2ESwa88y8QuUiGKW6ePdDSTt/view?usp=sharing.

Task. Implement a **biased random walk** to generate item sequences for Facebook pages, and create positive and negative training samples from the sequences. Train a word2vec (skip-gram) model to learn embeddings for graph nodes, and evaluate these embeddings qualitatively via examining the k-nearest neighbors of selected Facebook page queries.

Dataset. The dataset for this question is a page-page graph of verified Facebook sites. Nodes represent official Facebook pages while the links are mutual likes between sites. Node features are extracted from the site descriptions that the page owners created to summarize the purpose of the site. This graph was collected through the Facebook Graph API in November 2017 and restricted to pages from 4 categories which are defined by Facebook. These categories are: politicians, governmental organizations, television shows and companies. The task related to this dataset is multi-class node classification for the 4 site categories. The dataset is available at http://snap.stanford.edu/data/facebook_large.zip

Note. To match test cases, you may need to restart the runtime of your Colab and run the execution from scratch.

(a) [5 Points]

We provide a networkx graph constructed with the full dataset. Your task is to construct two smaller maximal subgraphs of degree k and higher, which will be useful in keeping the runtime of random walks and skip-gram training to a feasible range.

1. ***Small Graph*** with $k = 15$
2. Tiny Graph with $k = 30$

Hint 1: See documentation for networkx.k_core

Hint 2: Tiny Graph should have 804 nodes and 24266 edges.

Report the number of nodes and edges in the ***Small Graph***.

(b) [10 points]

Implement the **Biased Random Walk** of node2vec, which trades off between local and global neighborhood views of a node u via breadth-first search (BFS) and depth-first search (DFS) respectively. node2vec contains two parameters to interpolate between BFS and DFS:

1. Return Parameter (**p**): return back to previous node, i.e. immediately revisit a node in the walk

2. In-Out Parameter (**q**): ratio of moving outward (DFS) vs. inward (BFS). A high **q** value biases the walk to visiting local nodes, and a low **q** value towards distant nodes.

Recall the un-normalized random walk transition probabilities from Slide 43 in the lecture notes:

1. to return to previous node = $1/p$
2. to visit a local node = 1
3. to move forward = $1/q$

Instructions. Modify the provided code skeleton and implement and run a random walk sequence with $p = 1$, $q = 1$, iter= 5 and step= 10 on the ***Small Graph***, where **iter** is the number of iterations of random walks and **step** is the number of steps taken on a single random walk. Note that the Tiny Graph is provided only as a test case to help you debug your code. Submit the code snippet of your random walk (a single Colab cell) as a png or jpeg file.

Test Case: On Tiny Graph, the sequence of random walks should have length = 4020.

(c) [5 points]

We will now utilize the ***Small Graph*** random walk sequence to create positive and negative samples. We will learn embeddings for our Facebook pages by using these samples to train a skip-gram model with negative sampling (SGNS). In simple terms, a skip-gram model tries to predict the context words around a given center word. A more detailed overview of skip-gram modeling can be found at this blog post [2] and the Stanford CS 224D: Deep Learning for NLP Lecture Notes, though it is not required for this question. You will be provided **template code** to create positive and negative training examples from the random walk sequence, initialize the dataset and keras model, and plot the training loss. You will also be provided hyperparameters to train the model.

Report:

1. The training loss plot of your skip-gram model
2. The percentage drop in training loss after 10 epochs

Test Case: the output of *generate_examples()* in the **template code** on ***Small Graph*** should be:

```
Targets shape: (2498691,)
Contexts shape: (2498691,)
Labels shape: (2498691,)
Weights shape: (2498691,)
```

IMPORTANT: To match our output, you must ensure that the total length of the random walk sequence for a given node is n `num_steps`, including the starting node.

Hint: The total runtime of skip-gram training should be under 10 minutes on ***Small Graph***, and under 2 minutes on Tiny Graph.

(d) [10 Points]

Analyze the embeddings learnt with the random walk and skip-gram model by computing the cosine similarity of given Facebook page queries with all other embeddings learnt on the ***Small Graph***. You will be required to write this code within the provided skeleton.

Report the 5-nearest-neighbors (other than the node itself) for each of the following four nodes: "*Glee*", "*United States Air Force*", "*NASA's Curiosity Mars Rover*", and "*Barack Obama*".

Hint 1: don't forget to L2 normalize your embeddings! The L2 norm of the query vector for "*Brooklyn Nine Nine*" on Tiny Graph should be ≈ 1.37.

Hint 2: see documentation for tf.math.top_k

Test Case: On the provided Tiny Graph, the 5-nearest neighbors for the query "*Brooklyn Nine Nine*" are "*You The Jury*, "*MasterChef*, "*Houdini & Doyle*, "*Famous*, "*The Exorcist FOX*.

What to submit

(i) Upload your code to Gradescope as an *.ipynb* file. The code is to ensure that each person has written their own solution. We may choose to run your code to validate your solution but you do not have to worry about implementation details.

(ii) Report the number of nodes and edges after constructing the ***Small Graph***.

(iii) Submit the code snippet of your random walk sequence generation (single Colab cell provided in the **template code**).

(iv) Include the plot of skip-gram training accuracy and percentage drop after 10 epochs with embeddings learnt on ***Small Graph***.

(v) Include in your writeup the top 5 similar Facebook pages of the queries in part (d) with embeddings learnt on ***Small Graph***.

3 Data Streams (45 points)

In this question, we are going to follow an algorithm for determining the approximate frequencies of the unique items in a data stream. We will specifically investigate how we can

get a feasible approximation that uses less space than the naive solution but is still a good estimate of the actual frequencies. We will also experiment with a real stream dataset to empirically investigate our claims.

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 $\lceil \log \frac{1}{\delta} \rceil$ *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 each of the hash functions, and the count for the j-th hash function $c_{j,h_j(i)}$ is incremented by 1.

Note: You can assume that the hash functions are independent and totally random (see: https://courses.csail.mit.edu/6.851/spring12/scribe/lec10.pdf).

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) [4 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) [5 Points]

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

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

Hint: You can show that the inequality holds for $\tilde{F}[i]$ by showing that it holds for the minimum j or that it holds for all j where $c_{j,h_j(i)}$ denotes the count for the j-th hash function.

(c) [12 Points]

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

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

where, as mentioned, t is the length of the stream.

(d) [12 Points]

Prove that:

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

Hint 1: Use Markov inequality and the independence of hash functions.

Hint 2: Use the fact that $\widetilde{F}[i] = \min_j\{c_{j,h_j(i)}\}$ and thus $\widetilde{F}[i] \leq c_{j,h_j(i)} \forall j$.

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) [12 Points]

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

Dataset. The dataset in **streams/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 aforementioned 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 $E_r[i] = \frac{\tilde{F}[i]-F[i]}{F[i]}$ 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 be a scatter plot and 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 axis labels. 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 pseudocode), 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 **Algorithm** 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.

Sanity Check 1: On the tiny dataset, the actual word frequencies should be in range around 10^{-7} to 0.05 and the corresponding relative errors should be in range around 10^{-4} to 3×10^5.
Sanity Check 2: On the tiny dataset, nearly all the words with frequency roughly in range 10^{-3} to 10^{-4} have their relative errors below 10^{-1}.

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. An approximate condition on a word frequency to have a relative error below 1. [part (e)]

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

References

[1] Grover, A., and Leskovec, J. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining* (2016), pp. 855–864.

[2] McCormick, C. Word2vec tutorial - the skip-gram model. retrieved from http://www.mccormickml.com.