Homework 1 of CS 165A (Fall 2020)

University of California, Santa Barbara

Assigned on Oct 1, 2020 (Thursday)

Due at 11:59 pm on Oct 20, 2020 (Tuesday)

Notes:

- Be sure to read "Policy on Academic Integrity" on the course syllabus.
- Any updates or correction will be posted on the course Announcements page and piazza, so check there occasionally.
- You must do your own work independently. We will use software to automatically detect any plagiarisms.
- Please scan your answers and submit your homework via Gradescope. These includes a written part and code in the form of runnable Python 3 scripts (a template will be given for each question).
- TA in charge of this homework: Kaiqi Zhang (`kzhang70@ucsb.edu`)

Question 1. (14 points) Refreshers on Optimization and probability fundamentals.

Some standard notations in optimization: $\max, \min, \arg\max, \arg\min$ are commonly used when working with optimizations programs. The subscripts of these operators denote the "argument / variable" that you are optimizing over. For example, in $\min_{\theta \in \Theta} f(\theta)$, f is the objective function or criterion function, θ is the argument or variable that you are optimizing over. Θ is the domain that you can choose θ from (sometimes abbreviated when it is the whole space or clear from context). For example, $\min_{\theta \in \Theta} f(\theta)$ returns the minimum objective function value; $\arg\min_{\theta} f(\theta)$ returns the argument θ^* that achieves the minimum function value.

(a) (2 points) (Continuous optimization) Let $x_1, \ldots, x_n$ be real values. Consider a quadratic function $f(\theta) = \sum_{i=1}^{n} w_i (x_i - \theta)^2$. Assume $w_i > 0$. Derive the optimal solution θ^* that minimizes $f(\theta)$ - denoted by $\theta^* = \arg\min_{\theta} f(\theta)$? What happens if w_i is negative?

(b) (2 points) (counting and combinatorics) If $2n$ kids are randomly divided into two equal subgroups, find the probability that the two tallest kids will be: (i) in the same subgroup; (ii) in different subgroups.

(c) (2 points) (Bayes rule) In answering a question on a multiple choice test, a candidate either knows the answer with probability p ($0 \leq p < 1$) or does not know the answer with probability

$1-p$. If she knows the answer, she puts down the correct answer with probability 0.99, whereas if she guesses, the probability of his putting down the correct result is $1/k$ (k choices to the answer). Find the conditional probability that the candidate knew the answer to a question, given that she has made the correct answer.

(d) (2 points) (Likelihood and maximum likelihood) Consider a biased coin with the probability of turning up head $0 < p < 1$. We flip the coin 10 times and get a sequence of outcomes: $\{T, H, H, T, T, H, H, H, T, H\}$. We know that the probability (likelihood) of observing this sequence

$$L(p) := (1-p) \cdot p \cdot p \cdot (1-p) \cdot (1-p) \cdot p \cdot p \cdot p \cdot (1-p) \cdot p = p^6(1-p)^4$$

Calculate the value of p that maximizes the likelihood $L(p)$.

(Hint: you may wish to consider maximizing $\log L(p)$ instead. Why does it preserve the $\arg\max$ — the p^* that maximizes $L(p)$?)

(e) (2 points) (Calculus, gradients) Let $w \in \mathbb{R}^d$ be a column vector. Let $x_1, x_2, \ldots, x_n \in \mathbb{R}^d$ be column vectors of the same dimension d and $y_1, \ldots, y_n \in \mathbb{R}$ be scalars. Let $\lambda \in \mathbb{R}$ be a non-negative scalar value. Consider function $F(w) = \sum_{i=1}^n (x_i^T w - y_i)^2 + \lambda \|w\|^2$. Calculate the gradient of F. Recall that the gradient with respect to a vector of variables is the vector of partial derivatives with respect to each variable w_i:

$$\nabla f(w) = \left[\frac{\partial F(w)}{\partial w_1}, \ldots, \frac{\partial F(w)}{\partial w_d} \right]^T \in \mathbb{R}^d.$$

(Hint: It is easier to work out the partial derivatives one at a time, then concatenate the partial derivatives into a gradient by applying the definition above. The same hint also applies to other instances where you need to calculate the gradient.)

(f) (2 points) (Chain-rule and softmax function) Let $x_1, \ldots, x_n$ be real values. Let

$$f(x_1, \ldots, x_n) = \log \sum_{i=1}^n \exp(x_i).$$

This is called the softmax function or the log-sum-exp function[1] Calculate the gradient of f w.r.t. vector $x = (x_1, \ldots, x_n)^T$.

(g) (2 points) (Simple mathematical proof) For the soft-max function in part (f). Prove that $\max_i x_i \leq f(x_1, \ldots, x_n) \leq \max_i(x_i) + \log n$.

(Hint: A good practice when writing proof is to write a sequence of inequalities and explain every line in the following form:

"$f(x_1, \ldots, x_n) = \log \sum_{i=1}^n \exp(x_i) \leq \ldots \leq \ldots \leq \max_i(x_i) + \log n$. The first inequality is by definition, the second inequality is because ..., the third inequality is because ... ")

[1]Note that this softmax function is a scalar function $\mathbb{R}^n \to \mathbb{R}$ and it is different from the "softmax" transformation typically used in machine learning to convert any score vector to a probability vector, i.e., a vector valued function from $\mathbb{R}^n \to \mathbb{R}^n$. The latter is a misnomer but has a wide-spread misuse to the point that it becomes a convention. We will be working with the other function too. To not confuse ourselves, we call this the soft-argmax function. See more details in Q6.)

Question 2. (8 points) Refreshers on time complexity, data structure and python / numpy. The autograder uses python3.7, please write codes for python3.

(a) (2 points) In numpy, generate two matrices A and B with size 5 by 4 and size 4 by 3 respectively. In matrix A, make sure that the values are $1, 2, 3, \ldots, 19, 20$, from left-to-right then top to bottom. In matrix B, make sure that the values are $1, 2, 3, \ldots, 11, 12$ from top to bottom, then left to right. Write a script that print matrix A, matrix B and the matrix product of AB. Attach the output to your written submission, and also submit the code that generates these outputs with a filename "Q2a.py".

Hint: you can use numpy.dot for matrix multiply.

(b) (2 points) (Sparse matrix-vector multiplication) What is the worst case time complexity (in Big O notation) of multiplying a matrix A of dimension $\mathbb{R}^{n\times n}$ with a dense vector $v \in \mathbb{R}^n$? What if matrix A is sparse, denote the number of non-zero elements by $\text{nnz}(A)$? Write a python function that takes a nonnegative integer n and outputs a sparse matrix A of size $(n-1) \times n$, such that for any $x \in \mathbb{R}^n$, $Ax = [x_1 - x_2, ..., x_{n-1} - x_n]^T$. Write a script that calls this function and print the resulting A for $n = 5$. (Make sure you use the sparse matrix library in python is scipy.sparse.)

(c) (2 points) (Manipulating text using python, data structures) For the provided training data file, write a python function to find the unique words and print the corresponding frequency for the 20 most common words. Each line of training data contrains a list of words and a number. You can ignore the number for now. What are the data-structures you used and what is the time-complexity of your function in Big O notation. What is the space complexity of your code for this task? Again, attach your code with the function and the corresponding script.

Hint: among the build-in data structures, eg., list, dict, set, etc., which one best suit this problem? The package "collections" may be helpful as well.

(d) (2 points) (Simple recursion in python) Implement a recursive python function with name "factorial" that takes a positive integer n and output its "factorial" $n! = 1 \cdot 2 \cdot \ldots \cdot n$.

Question 3. (10 points) Problem solving agents, rationality and descriptions of the environment.

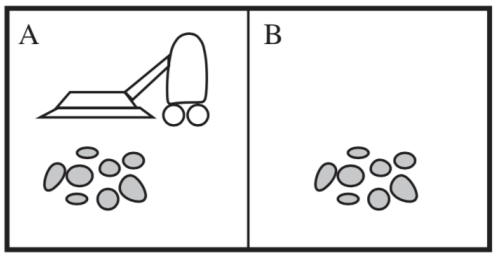

Figure 2.2 A vacuum-cleaner world with just two locations.

Percept sequence	Action
$[A, Clean]$	*Right*
$[A, Dirty]$	*Suck*
$[B, Clean]$	*Left*
$[B, Dirty]$	*Suck*
$[A, Clean], [A, Clean]$	*Right*
$[A, Clean], [A, Dirty]$	*Suck*
⋮	⋮
$[A, Clean], [A, Clean], [A, Clean]$	*Right*
$[A, Clean], [A, Clean], [A, Dirty]$	*Suck*
⋮	⋮

Figure 2.3 Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 2.2.

(a) Let us inspect the vacuum-cleaner example again from the textbook. Under the assumption listed on textbook page 38:

(i) (2 points) Explain that why this vacuum-cleaner agent function described in Figure 2.3 is rational.

(ii) (2 points) If each movement of the cleaner generate a unit cost, explain that now a rational agent needs to maintain an internal state.

(iii) (2 points) Back to the original problem, now there is a naughty pet dog in the environment. At each time step, each clean square has a 50% chance of becoming dirty. Briefly explain how can you modified the rational agent in this case.

(b) For each of the following activities, give a **PEAS** description of the task environment and characterize it in terms of the properties listed in textbook Section 2.3.2.

(i) (2 points) Playing soccer as a robot.

(ii) (2 points) Internet laptop-shopping agent. (an intelligent agent that helps a customer to shop for a laptop on a website.)

Please be concise in your answers.

Author categorization with machine learning. You will be given an authorship attribution corpus where each text instance is extracted from works by authors of Victorian Era. Each text instance consists of around 1,000 words and can be associated with an integer label indicating different authors from 1 to 15.

The following three questions will guide you through a simple pipeline that creates a multi-class classification agent using machine learning. You will be writing codes to actually build everything from scratch.

The provided dataset consists of two files. "training.txt"' contains the training data. A machine learning algorithm should only use the training data for training. Then there is another dataset "testing.txt" that contains the testing data that is used exclusively for evaluating the performance of your trained classifier.

Each of these files will have an arbitrary number of lines and each line represents one text instance (also referred to as document). Each line will have the format **"D,L"**, where L is the class (label) of the document (from 1 to 15) and D is the text of the document. Note that D might contain several sequences of words and **no punctuations are included**. Here is an example of a line, with most words replaced by ellipsis (...) for reading convenience:

```
wish for solitude he was twenty years of age ... of her brown legs was
entirely uncovered ,1
```

where the label of the document is 1 (author #1). And here is an example of a text instance from another author (author #5):

```
he owes nothing the company in which ... from his literary proprietor by
the of his guineas ,5
```

As we discussed in the lecture, a typical modern AI pipeline contains three components **modeling**, **learning** and **inference**. In the context of this problem, the part of **modeling** that we need to do is only **feature extraction**.

You will implement feature extractors in Q4, and then implement the learning and inference components in Q6 of a logistic regression classifier. We will also talk about a data-driven way of selecting hyperparameters by **cross validation** in Q5.

Question 4. (12 points) (Feature extraction) The goal of feature extraction is to transform the raw data describing a document — a string of text — into a numerical vector of floating point numbers (in numpy.array format), which can then be used by a machine learning model.

In this assignment, you are given raw files where each line contains a string of the format "word1 word2 word3 ... wordN, digit", where "word1 word2 word3 ... wordN" is the input or x and digit is the label or y. One example is that "Tom chases Jerry , 15". You are expected to train a machine learning model to consume the input "Tom chases Jerry" and predict the label "15".

The first step is to do feature extraction on the input document (a string of text) and generate a vector of numerical numbers can be later processed by machine learning models. You will implement two popular feature extractors in this question.

(a) (6 points) The Bag-of-Word feature is a popular feature extractor, which counts the number of each word in the input string. In particular, we may first generate a vocabulary containing all words and then count how many times each word appears in the input string.

For example, given a list of string ["A great game", "The election was over", "Very clean match", "A clean but forgettable game", "It was a close election"], we may first generate a vocabulary [a, great, game, the, election, was, over, very, clean, match, but, forgettable, it, close]. Then, given a new input string "A very close game", we can generate the bag-of-word feature as [1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1]. Note that the order matters in the bag-of-word.

Please complete the python function "bag_of_word_feature" in "Q4.py". You should use what you have implemented in Q2(c) to generate the vocabulary and a dictionary that maps the words in the vocabulary into their index. This dictionary should be used as a global variable when defining the "bag_of_word_feature".

(b) (6 points) A slightly improved feature extractor is the TF-IDF [3, 6], which stands for term frequency-inverse document frequency. Term frequency is defined as

$$\text{TF}(t) = \frac{\text{Number of times term t appears in a document}}{\text{Total number of terms in the document}}.$$

Inverse document frequency is defined by

$$\text{IDF}(t) = \log_{10}\left(\frac{\text{Total number of documents}}{\text{Number of documents with term t in it}}\right).$$

TF-IDF is defined as the product of TF and IDF

$$\text{TF-IDF}(t) = \text{TF}(t) \cdot \text{IDF}(t).$$

TF-IDF feature vector of a document enumerates the TF-IDF for all possible "term" t in the vocabulary, and can be represented by a sparse vector in $\mathbb{R}^{\text{size of the vocabulary}}$. The vector is non-zero only for those coordinates that represent a "word" that appeared in this document.

The following is an example of tf-idf. Consider a document containing 100 words wherein the term "cat" appears 3 times. The term frequency (i.e., tf) for cat is then (3 / 100) = 0.03. Now, assume we have 10 million documents and the word cat appears in one thousand of these. Then, the inverse document frequency (i.e., idf) is calculated as log(10,000,000 / 1,000) = 4. Thus, the Tf-idf weight is the product of these quantities: 0.03 * 4 = 0.12.

Please complete the python function "TF_IDF_feature(document)" in Q4.py that extracts the TF-IDF feature vector of a given document. the output should be a scipy.sparse vector of dimension $d \times 1$ where d is the size of the vocabulary.

Notice that TF-IDF does not only depend on the single document alone, but rather uses information from the entire collection of documents (a corpus) too. In our case, the corpus is all documents in "train.txt".

Think about what are these information needed and how to represent them. You should avoid recomputing these information each time "TF_IDF_feature" is called. In other word, you should cache / pre-compute those information that is needed ahead of time and keep them as global variables that "TF_IDF_feature" when it is defined.

Hint:

1. Can you implement TF by calling "Bag_of_word_ feature"?

2. Pay attention to corner cases: What if the input document contains words that are not in the vocabulary? Simply ignoring it would be a good choice, but you need to make sure that the code does not break.

Question 5. (12 points) (Hold-out data and cross validation) The next step is to build some tool for evaluating the performance of a classifier learned by a machine learning algorithm.

Let h be a classifier. More formally, $h : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ is the feature space and $\mathcal{Y}$ is the label space. And we denote the space of all classifiers to be $\mathcal{H}$.

The most natural performance metric is the classification error, which measures the expected error rate. The expectation is taken over the distribution of the data under the typical assumption that the data points $(x_1, y_1), ..., (x_n, y_n)$ are drawn i.i.d. from some distribution $\mathcal{D}$ defined on $\mathcal{X} \times \mathcal{Y}$.

$$\text{Err}(h) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[\mathbf{1}(h(x) \neq y)]$$

where $\mathbf{1}(\cdot)$ is the indicator function that outputs 1 is the condition is true and 0 otherwise.

We can also define the error that we calculate on a dataset. Specifically, let $(x_1, y_1), ..., (x_n, y_n)$ be the dataset used for training, we denote the empirical error on this data set as

$$\hat{\text{Err}}(h) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}(h(x) \neq y).$$

Furthermore, we define the generalization error to be $\text{GenErr} = \max_{h\in\mathcal{H}} |Err(h) - \hat{Err}(h)|$ — the difference between the training error and the expected error on a new data point.

(a) (4 points) Let $\hat{h}$ be the learned classifier and h^* be the optimal classifier., i.e.

$$\hat{h} = \arg\min_h \hat{\text{Err}}(h), \quad h^* = \arg\min_h \text{Err}(h)$$

Prove that:

$$\text{Err}(\hat{h}) \leq \text{Err}(h^*) + 2\text{GenErr}.$$

(b) (4 points) In practice, we need to evaluate a classifier on a hold-out dataset. We randomly partition the data into a training data set and a validation dataset. Show that

$$\mathbb{E}[\hat{\text{Err}}_{\text{Val}}(\hat{h})] = \mathbb{E}[\text{Err}(\hat{h})],$$

where $\hat{\text{Err}}_{Val}$ denotes the empirical error calculated on the validation set, and the expectation is averaging over all random variables including the training data of $\hat{h}$.

(c) (4 points) Implement a function that takes a list of predicted labels and a list of ground truth labels and output the error rate.

(d) (Bonus, 4 points) There is something even better than hold-out for evaluating a classifier's performance, called K-fold Cross-Validation. The idea is that we split the data into K random partitions. We use all combinations of (K-1) folds for training and the remaining one fold for validation. At the end of the day, we take the average of the K validation errors that we obtain as the cross validation error. It is clear based on the result in (b) that the the cross-validation error is an unbiased estimator of the expected error of the classifier trained on an i.i.d. data set of size $(K-1)n/K$ (assuming that n/K is an integer).

Cross-validation is generally used for hyper-parameter tuning. Suppose we have a hyper-parameter λ in our machine learning model with a sequence of candidate values $[\lambda_1, \lambda_2, \lambda_3]$. We may set λ to each candidate value and compute the cross validation error. Finally, we will select the λ with the lowest cross validation error.

However, simple as it is, cross-validation is easy to misuse, especially when there are data-preprocessing steps that are needed before training a classifier. For instance, in this example of text classification, the feature extraction step requires constructing a vocabulary as well as some normalization steps (if TF-IDF is used). Discuss what can go wrong if these feature extractors are created based on all data, rather than just the training data.

Question 6. (40 points) (Implement multi-class logistic regression classifier) We learned about binary logistic regression in the lectures. In this question, we will take the TF-IDF features vector from the previous question and train a multiclass logistic regression classifier.

Logistic regression classifier is a popular machine learning model for predicting the probability of individual labels. It has a parameter matrix $\Theta \in \mathbb{R}^{d \times k}$, where k is the number of classes (in this case $k = 15$) and d is the dimension of the feature vector. These are the "free parameters" in the logistic regression. Each specific choice of Θ corresponds to a classifier. We will denote $\Theta = [\theta_1, \theta_2, ..., \theta_k]$ where $\theta_i \in \mathbb{R}^d$ represents the ith column of the parameter matrix Θ.

Recall the *modeling-learning-inference* paradigm from the lecture, by stating the feature extractor being TF-IDF and the classifier family being linear logistic regression classifiers, we have already completed the **modelling** part of the design.

Learning amounts to finding a specific parameter Θ that has low classification error. **Inference** is about using this classifier to predict the label.

Given a feature vector $\mathbf{x} = (x_1, x_2, ..., x_d)^T$, the prediction that generates by a logistic regression classifier:

$$\hat{y} = \text{soft\_argmax}(\Theta^T \mathbf{x}),$$

where soft_argmax : $\mathbb{R}^d \to \mathbb{R}^k$ is a vector-valued function that outputs $\text{soft\_argmax}(z)_i = e^{z_i} / \sum_{j=1}^{k} e^{z_j}$ for $i = 1, 2, ..., k$. Please convince yourself that the output vector from soft_argmax is always a probability distribution.

The predicted label of a logistic regression classifier is then taken as $\arg\max_{j \in \{1,2,...,k\}} \{\hat{y}_j\}$.

While ultimately we will evaluate the classifier using the classification error on the hold-out data. These quantities involving indicators are not differentiable thus not easy to optimize.

The training of a logistic regression classifier thus involves minimizing a surrogate loss function which is continuously differentiable. In particular, the multi-class logistic loss function for a data point (x, y) that we use is the following

$$L(\Theta, (\mathbf{x}, y)) := -\sum_{j=1}^{k} y[j] \log(\hat{y}[j]).$$

This is also called the Cross-Entropy loss.

Here, we used the one-hot encoding of the label y. If the true label is 5 (the fifth author wrote this piece), $y \in \mathbb{R}^k$ is a vector with only the 5th element being 1 and all other elements are zero.

Finally, the continuous optimization problem that we solve to "train" a logistic regression classifier using $(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)$ is

$$\min_{\Theta} \sum_{i=1}^{n} L(\Theta, (\mathbf{x}_i, y_i)) + \lambda \|\Theta\|_F^2. \tag{1}$$

The Frobenius norm $\|\Theta\|_F = \sqrt{\sum_{i,j} \Theta[i,j]^2}$.

(a) (8 points) Implement a python function with name "inference" that takes Θ and $\mathbf{x}$ as inputs and outputs $\log(\hat{y})$.

(b) (8 points) Derive the gradient of the cross-entropy loss with respect to Θ. Implement a python function with name "gradient" that takes Θ, $\mathbf{x}$ and y as input and outputs $\nabla_\Theta L(\Theta, (\mathbf{x}, y))$.

(Hint: The gradient has the same dimension as Θ. You may calculate the partial gradient w.r.t. each each column θ_i. Lastly, representing the gradient as a scipy.sparse matrix will make it more efficient.)

(c) (8 points) Derive the gradient of (1). Also, come up with a way to construct a stochastic gradient using a randomly chosen data point. Implement two functions "full_gradient" and "stochastic_gradient". The "full_gradient" takes the Θ, and the full dataset as inputs, and outputs the gradient of the overally objective (1) at Θ. Think about how do we represent the full dataset.

The "stochastic_gradient" takes the same input, but outputs an unbiased estimate of the true gradient.

What are the time complexity of these two functions?

(Hint: 1.numpy.random.int could be useful. 2. Eqn (1) is a sum of cross-entropy loss functions, rather than the average.)

(d) (8 points) Implement both the gradient descent algorithm and the stochastic gradient descent algorithm.

Specifically, implement a python function with name "train". "train" takes an initial parameter Θ, and a gradient function (this could be either "full_gradient" or "stochastic_gradient"), a learning rate parameter and the number of iterations, the output will be the updated parameter Θ.

(e) (8 points) Write a python script that drives the training process using GD and SGD. Set $\lambda = 1$ and run GD for 1000 iterations and SGD for 10 passes of the data worth of iterations. In other word, for SGD, repeat SGD step for $10 \times N$ where N is the size of training set. Compare GD and SGD by plotting the errors on the training set against the the number of iterations and also (in a separate figure, against) real-clock time. Do the same for the errors on the test data set.

You could use what you wrote in Q5(c) to calculate the error. Calculate the error every once 100 iterations for GD and every 1 data pass for SGD.

Make sure that you start with a fixed random seed using `numpy.random.seed(93106)` so we can replicate your results.

(Hint: the updates should be quite efficient if you use scipy.sparse. Why is the stochastic gradient is sparse (no need to answer this question in your solution, but please think about it)?)

(f) (Bonus 5 points) Plot the validation error as a function of lambda when you choose

$$\lambda \in \{0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000\}.$$

For each experiment, run SGD for 10 data passes. Please use logarithmic scale in both the x and y axes.

(g) (Bonus 3 points) What is the prediction accuracy you are getting? What are some ideas that you have in further improving the accuracy? (no need to implement anything, just think outside the box)

(h) (Bonus 3 points) How would you interpret the learned coefficient vectors $\theta_1, ..., \theta_k$? Plotting the "words" corresponding to the largest 10 coordinates of θ_i for $i = 1, 4, 8$.

You may pass the coefficient vector (make sure you remove negative ones) along with the (correctly ordered) vocabulary list to `https://www.wordclouds.com/` and generate some interesting visualization.