CIS519: Applied Machine Learning	**Fall 2019**
Homework 2	
Handed Out: October 7, 2019	*Due: October 21, 2019 at 11:59pm*

Version 2

- Feel free to talk to other members of the class in doing the homework. I am more concerned that you learn how to solve the problem than that you demonstrate that you solved it entirely on your own. You should, however, **write down your solution yourself**. Please include at the top of your document the list of people you consulted with in the course of working on the homework.
- While we encourage discussion within and outside the class, cheating and copying code is strictly not allowed. Copied code will result in the entire assignment being discarded at the very least.
- Please use Piazza if you have questions about the homework. Also, please come to the TAs recitations and to the office hours.
- Handwritten solutions are not allowed. All solutions must be typeset in Latex. Consult the class' website if you need guidance on using Latex. If you don't have a lot of experience with Latex (or even if you do), we recommend using Overleaf (`https://www.overleaf.com`) to write your solutions. You will submit your solutions as a single pdf file (in addition to the package with your code; see instructions in the body of the assignment).
- The homework is due at 11:59 PM on the due date. We will be using Gradescope for collecting the homework assignments. You should have been automatically added to Gradescope. If not, please ask a TA for assistance. Please do **not** hand in a hard copy of your write-up. Post on Piazza and contact the TAs if you are having technical difficulties in submitting the assignment.
- Here are some resources you will need for this assignment
  - `https://www.seas.upenn.edu/~cis519/fall2019/assets/HW/HW2/hw2-materials.zip`

1 Overview

In this homework assignment, you will experiment with several different linear classifiers and analyze their performances both real and synthetic datasets. The goal is to understand the differences and similarities between the algorithms and the impact that the dataset characteristics have on the algorithms' learning behaviors and performances.

In total, there are seven different learning algorithms which you will implement. Six are variants of the Perceptron algorithm and the seventh is a support vector machine (SVM). The details of these models is described in Section 2.

In order to evaluate the performances of these models, you will use several different datasets. The first two datasets are synthetic datasets that have features and labels that were programatically generated. They were generated using the same script but use different input parameters that produced sparse and dense variants. The second two datasets are for the task of named-entity recognition (NER), identifying the names of people, locations, and organizations within text. One comes from news text and the other from a corpus of emails. For these two datasets, you need to implement the feature extraction yourself. All of the datasets and feature extraction information are described in Section 3.

Finally, you will run two sets of experiments, one on the synthetic data and one on the NER data. The first set will analyze how the amount of training data impacts model

performance. The second will look at the consequences of having training and testing data that come from different domains. The details of the experiments are described in Section 4.

2 Algorithms

This section details the seven different algorithms that you will use in the experiments. For each of the algorithms, we describe the initialization you should use to start training and the different parameter settings that you should use for the experiment on the synthetic datasets. Each of the update functions for the Perceptron, Winnow, and Perceptron with AdaGrad will be unittested on Gradescope, so please do not edit the function definitions.

2.1 Perceptron Variants

The Perceptron is an online mistake-driven algorithm. We will consider three different variations of the percepron – the basic Perceptron, Winnow, and Adagrad – and for each one, you will implement the standard and averaged versions.

Since the code for the basic Perceptron is provided, implementing Winnow and Adagrad will require minor modifications to the update rule. Similarly, once you have implemented the averaged basic Perceptron, the averaged versions of the other two algorithms should be straightforward.

Note that we make a distinction in this section between the **parameters** and the **hyperparameters** of a model. The parameters are those which are learned (e.g., you update them according to gradient descent), whereas the hyperparameters are properties of the model or learning algorithm (e.g., the depth of the decision tree, the learning rate for stochastic gradient descent).

a. **Basic Perceptron:** This is the basic version of the Perceptron Algorithm. In this version, an update will be performed on the example $(\mathbf{x}, y)$ if $y(\mathbf{w}^\intercal \mathbf{x} + \theta) \leq 0$.

The Perceptron needs to learn both the bias term θ and the weight vector $\mathbf{w}$ parameters. When the Perceptron makes a mistake on the example $(\mathbf{x}, y)$, both $\mathbf{w}$ and θ need to be updated using the following update equations:

$$\mathbf{w}^{\text{new}} \leftarrow \mathbf{w} + \eta \cdot y \cdot \mathbf{x}$$
$$\theta^{\text{new}} \leftarrow \theta + \eta \cdot y$$

where η is the learning rate.

Hyperarameters: Because η does not have any effect, we set $\eta = 1$.[1] There are no hyperparameters to tune for this model.

[1]If we assume that the order of the examples presented to the algorithm is fixed, we initialize $\mathbf{w} = \mathbf{0}$ and $\theta = 0$, and train both together, then the learning rate η does not have any effect. In fact you can show that, if $\mathbf{w}_1$ and θ_1 are the outputs of the Perceptron algorithm with learning rate η_1, then $\mathbf{w}_1/\eta_1$ and θ_1/η_1 will be the result of the Perceptron with learning rate 1 (note that these two hyperplanes give identical predictions).

Initialization: $\mathbf{w} = [0, 0, \ldots, 0]$ and $\theta = 0$.

b. **Winnow:** The Winnow algorithm is a variant of the Perceptron algorithm with multiplicative updates. Since the algorithm requires that the target function is monotonic, you will only use it on the synthetic datasets.

The Winnow algorithm only learns parameters $\mathbf{w}$. We will fix $\theta = -n$, where n is the number of features. When the Winnow algorithm makes a mistake on the example $(\mathbf{x}, y)$, the parameters are updated with the following equation:

$$w_i^{\text{new}} \leftarrow w_i \cdot \alpha^{y \cdot x_i} \tag{1}$$

where w_i and x_i are the ith components of the corresponding vectors. Here, α is a promotion/demotion hyperparameter.

Hyperparameters: Choose $\alpha \in \{1.1, 1.01, 1.005, 1.0005, 1.0001\}$.

Initialization: $\mathbf{w} = [1, 1, \ldots, 1]$ and $\theta = -n$ (constant).

c. **Perceptron with AdaGrad:** AdaGrad is a variant of the Perceptron algorithm that adapts the learning rate for each parameter based on historical information. The idea is that frequently changing features get smaller learning rates and stable features higher ones.

To derive the update equations for this model, we first need to start with the loss function. Instead of using the hinge loss with the elbow at 0 (like the basic Perceptron does), we will instead use the standard hinge loss with the elbow at 1:

$$\mathcal{L}(\mathbf{x}, y, \mathbf{w}, \theta) = \max\{0, 1 - y(\mathbf{w}^\intercal \mathbf{x} + \theta)\} \tag{2}$$

Then, by taking the partial derivative of $\mathcal{L}$ with respect to $\mathbf{w}$ and θ, we can derive the respective graidents (make sure you understand how you could derive these gradients on your own):

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \begin{cases} \mathbf{0} & \text{if } y(\mathbf{w}^\intercal \mathbf{x} + \theta) > 1 \\ -y \cdot \mathbf{x} & \text{otherwise} \end{cases} \tag{3}$$

$$\frac{\partial \mathcal{L}}{\partial \theta} = \begin{cases} 0 & \text{if } y(\mathbf{w}^\intercal \mathbf{x} + \theta) > 1 \\ -y & \text{otherwise} \end{cases} \tag{4}$$

Then for each parameter, we will keep track of the sum of the parameters' squared gradients. In the following equations, the k superscript refers to the kth non-zero gradient (i.e., the kth weight vector/misclassified example) and t is the number of mistakes seen thus far.

$$G_j^t = \sum_{k=1}^{t} \left(\frac{\partial \mathcal{L}}{\partial w_j^k} \right)^2 \tag{5}$$

$$H^t = \sum_{k=1}^{t} \left(\frac{\partial \mathcal{L}}{\partial \theta^k} \right)^2 \tag{6}$$

For example, on the 3rd mistake ($t = 3$), G_j^3 is the sum of the squares of the first three non-zero gradients ($\left(\frac{\partial \mathcal{L}}{\partial w_j^1}\right)^2$, $\left(\frac{\partial \mathcal{L}}{\partial w_j^2}\right)^2$, and $\left(\frac{\partial \mathcal{L}}{\partial w_j^3}\right)^2$). Then $\mathbf{G}^3$ is used to calculate the 4th value of the weight vector as follows. On example $(\mathbf{x}, y)$, if $y(\mathbf{w}^\intercal \mathbf{x} + \theta) \leq 1$, then the parameters are updated with the following equations:

$$\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t + \eta \cdot \frac{y \cdot \mathbf{x}}{\sqrt{\mathbf{G}^t}} \tag{7}$$

$$\theta^{t+1} \leftarrow \theta^t + \eta \frac{y}{\sqrt{H^t}} \tag{8}$$

Note that, although we use the hinge loss with the elbow at 1 for training, you still make the prediction based on whether or not $y(\mathbf{w}^\intercal \mathbf{x} + \theta) \leq 0$ during testing.

Hyperparameters: Choose $\eta \in \{1.5, 0.25, 0.03, 0.005, 0.001\}$

Initialization: $\mathbf{w} = [0, 0, \dots, 0]$ and $\theta = 0$.

d. **Averaged Perceptrons:** You will also implement the averaged version of the previous three algorithms.

During the course of training, each of the above algorithms will have $K+1$ different parameter settings for the K different updates it will make during training. The regular implementation of these algorithms uses the parameter values after the Kth update as the final ones. Instead, the averaged version use the weighted average of the $K+1$ parameter values as the final parameter values. Let m_k denote the number of correctly classified examples by the kth parameter values and M the total number of correctly classified examples. The final parameter values are

$$M = \sum_{k=1}^{K+1} m_k \tag{9}$$

$$\mathbf{w} \leftarrow \frac{1}{M} \sum_{k=1}^{K+1} m_k \cdot \mathbf{w}^k \tag{10}$$

$$\theta \leftarrow \frac{1}{M} \sum_{k=1}^{K+1} m_k \cdot \theta^k \tag{11}$$

$$\tag{12}$$

For each of the averaged versions of Perceptron, Winnow, and AdaGrad, use the same hyperparameters and initialization as before.

Implementation Note: Implementing the averaged variants of these algorithms can be tricky. While the final parameter values are based on the sum of K different vectors, there is no need to maintain *all* of these parameters. Instead, you should implement these algorithms by keeping only two vectors, one which maintains the cumulative sum and the current one.

Additionally, there are two ways of keeping track of these two vectors. One is more straightforward but prohibitively slow. The second requires some algebra to derive but is significantly faster to run. Try to analyze how the final weight vector is a function of the intermediate updates and their corresponding weights. It should take less than a minute or two for ten iterations for any of the averaged algorithms. **You need to think about how to efficiently implement the averaged algorithms yourself.**

Further, the implementation for Winnow is slightly more complicated than the other two, so if you consistently have low accuracy for the averaged Winnow, take a closer look at the derivation.

2.2 Support Vector Machines

Although we have not yet covered SVMs in class, you can still train them using the `sklearn` library. We will use a soft margin SVM for non-linearly separable data. You should use the `sklearn` implementation as follows:

```
from sklearn.svm import LinearSVC
classifier = LinearSVC(loss='hinge')
classifier.fit(X, y)
```

`sklearn` requires a different feature representation than what we use for the Perceptron models. The provided Python template code demonstrates how to convert to the require representation.

Given training samples $S = \{(\mathbf{x}^1, y^1), (\mathbf{x}^2, y^2), \ldots, (\mathbf{x}^m, y^m)\}$, the objective for the SVM is the following:

$$\min_{\mathbf{w},b,\boldsymbol{\xi}} \frac{1}{2}||\mathbf{w}||_2^2 + C\sum_{i=1}^{m} \xi_i \tag{13}$$

subject to the following constraints:

$$y^i(\mathbf{w}^\intercal \mathbf{x}^i + b) \geq 1 - \xi_i \quad \text{for } i = 1, 2, \ldots, m \tag{14}$$

$$\xi_i \geq 0 \quad \text{for } i = 1, 2, \ldots, m \tag{15}$$

3 Datasets

In this section, we describe the synthetic and NER datasets that you will use for your experiments. For the NER datasets, there is also an explanation for the features which you need to extract from the data.

3.1 Synthetic Data

The synthetic datasets have features and labels which are automatically generated from a python script. Each instance will have n binary features and are labeled according to a l-of-m-of-n Boolean function. Specifically, there is a set of m features such that an example if positive if and only if at least l of these m features are active. The set of m features is the same for the dataset (i.e., it is not a separate set of m features for each individual instance).

We provide two versions of the synthetic dataset called sparse and dense. For both datasets, we set $l = 10$ and $m = 20$. We set $n = 200$ for the sparse data and $n = 40$ for the dense data. Additionally, we add noise to the data as follows: With probability 0.05 the label assigned by the function is changed and with probability 0.001 each feature value is changed. Consequently, the data is not linearly separable.

We have provided you with three data splits for both sparse and dense with 50,000 training, 10,000 development, and 10,000 testing examples. Section 4.1 describes the experiments that you need to run on these datasets.

3.1.1 Feature Representation

The features of the synthetic data provided are vectors of 0s and 1s. Storing these large matrices requires lots of memory so we use a sparse representation that stores them as dictionaries instead. For example, the vector `[0,1,0,0,0,1]` can be stored as `{"x2":1,"x6":1}` (using 1-based indexing). We have provided you with the code for parsing and converting the data to this format. You can use these for the all algorithms you develop except the SVM. Since you will be using the implementation of SVM from sklearn, you will need to provide a vector to it. You can use `sklearn.feature_extraction.DictVectorizer` for converting feature-value dictionaries to vectors.

3.2 NER Data

In addition to the synthetic data, we have provided you two datasets for the task of named-entity recognition (NER). The goal is to identify whether strings in text represent names of people, organizations, or locations. An example instance looks like the following:

```
[PER Wolff] , currently a journalist in [LOC Argentina] , played with
[PER del Bosque] in the final years of the seventies in
[ORG Real Madrid] .
```

In this problem, we will simplify the task to identifying whether a string is named entity or not (that is, you don't have to say which type of entity it is). For each token in the input, we will use the tag `I` to denote that token is an entity and `O` otherwise. For example, the full tagging for the above instace is as follows:

```
[I Wolff] [O ,] [O currently] [a] [O journalist] [O in] [I Argentina]
[O ,] [O played] [O with] [I del] [I Bosque] [O in] [O the] [O final]
[O years] [O of] [O the] [O seventies] [O in] [I Real] [I Madrid] .
```

Given a sentence $S = w_1, w_2, \ldots, w_n$, you need to predict the {`I`, `O`} tag for each word in the sentence. That is, you will produce the sequence $Y = y_1, y_2, \ldots, y_n$ where $y_i \in$ {`I`, `O`}

3.2.1 Datasets: CoNLL & Enron

We have provided two datasets, the CoNLL dataset which is text from news articles, and Enron, a corpus of emails. The files contain one word and one tag per line. For CoNLL, there are training, development, and testing files, whereas Enron only has a test dataset. There

are 14,987 training sentences (204,567 words), 336 development sentences (3,779 words), and 303 testing sentences (3,880 words) in CoNLL. For Enron there are 368 sentences (11,852 words).

Please note that the CoNLL dataset is available only for the purposes of this assignment. It is copyrighted, and you are granted access because you are a Penn student, but please delete it when you are done with the homework.

3.2.2 Feature Extraction

The NER data is provided as raw text, and you are required to extract features for the classifier. In this assignment, we will only consider binary features based on the context of the word that is supposed to be tagged.

Assume that there are V unique words in the dataset and each word has been assigned a unique ID which is a number $\{1, 2, \ldots, V\}$. Further, w_{-k} and w_{+k} indicate the kth word before and after the target word. The feature templates that you should use to generate features are as follows:

Template	Number of Features
w_{-3}	V
w_{-2}	V
w_{-1}	V
w_{+1}	V
w_{+2}	V
w_{+3}	V
w_{-1} & w_{-2}	$V \times V$
w_{+1} & w_{+2}	$V \times V$
w_{-1} & w_{+1}	$V \times V$

Each feature template corresponds to a set of features that you will compute (similar to the features you generated in problem 2 from the first homework assignment). The w_{-3} feature template corresponds to V features where the ith feature is 1 if the third word to the left of the target word has ID i. The $w_{-1}\&w_{+1}$ feature template corresponds to $V \times V$ features where there is one feature for every unique pair words. For example, feature $(i-1) \times V + j$ is a binary feature that is 1 if the word 1 to the left of the target has ID i and the first word to the right of the target has ID j. In practice, you will not need to keep track of the feature IDs. Instead, each feature will be given a name such as "$w_{-1} = \text{the}\&w_{+1} = \text{cat}$".

In total, all of the above feature templates correspond to a very large number of features. However, for each word, there will be exactly 9 features which are active (non-zero), so the feature vector is quite sparse. You will represent this as a dictionary which maps from the feature name to the value. In the provided Python template, we have implemented a couple of the features for you to demonstrate how to compute them and what the naming scheme should look like.

In order to deal with the first two words and the last two words in a sentence, we will add special symbol "SSS" and "EEE" to the vocabulary to represent the words before the first word and the words after the last word. Notice that in the test data you may encounter a word that was not observed in training, and therefore is not in your dictionary. In this

case, you cannot generate a feature for it, resulting in less than 7 active features in some of the test examples.

4 Experiments

You will run two sets of experiments, one using the synthetic data and one using the NER data.

4.1 Synthetic Experiments

This experiment will explore the impact that the amount of training data has on model performance. First, you will do hyperparameter tuning for Winnow and Perceptron with AdaGrad (both standard and averaged versions). Then you will generate learning curves that will plot the size of the training data against the performance. Finally, for each of the models trained on all of the training data, you will report test scores.

You should use accuracy to compute the performance of the model.

4.1.1 Parameter Tuning

For both the Winnow and Perceptron with AdaGrad (standard and averaged), there are hyperparameters that you need to choose. (The same is true for SVM, but you should only use the default settings.) Similarly to cross-validation from Homework 1, we will estimate how well each model will do on the true test data using the development dataset (we will not run cross-validation), and choose the hyperparameter settings based on these results.

For each hyperparameter value in Section 2, train a model using that value on the training data and compute the accuracy on the development dataset. Each model should be trained for 10 iterations (i.e., 10 passes over the entire dataset).

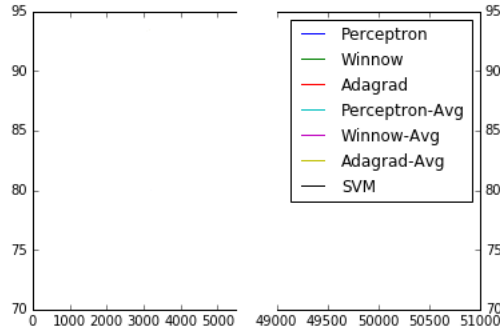
Make a table with the hyperparameter values and the corresponding accuracies. Repeat this for both the sparse and dense data.

4.1.2 Learning Curves

Next, you will train all 7 models with different amounts of training data. For Winnow and Perceptron with AdaGrad (standard and averaged), use the best hyperparameters from the parameter tuning experiment.

Each of the datasets contains 50,000 training examples. You will train each model 11 times on varying amounts of training data. The first 10 will increase by 500 examples: 500, 1k, 1.5k, 2k, ..., 5k. The 11th model should use all 50k examples. Each Perceptron-based model should be trained for 10 iterations (e.g., 10 passes over the total number of training examples available to that model). The SVM can be run until convergence with the default parameters.

For each model, compute the accuracy on the development dataset and plot the results using the provided code. There should be a separate plot for the sparse and dense data.



Answer the following questions:

1. Discuss the trends that you see when comparing the standard version of an algorithm to the averaged version (e.g., Winnow versus Averaged Winnow). Is there an observable trend?
2. We provided you with 50,000 training examples. Were all 50,000 necessary to achieve the best performance for each classifier? If not, how many were necessary? (Rough estimates, no exact numbers required)

4.1.3 Final Evaluation

Finally, for each of the 7 models, train the models on all of the training data and compute the accuracy on the actual test dataset. For Winnow and Perceptron with AdaGrad, use the best hyperparameter settings you found. Report these accuracies in a table.

4.1.4 [Extra Credit] Robustness to Noise

Included in the resources for this homework assignment is the code that we used to generate the synthetic data. We used a small amount of noise to create the dataset which you ran the experiments on. For extra credit, vary the amount of noise in either/both of the label and features. Then, plot the models' performances as a function of the amount of noise. Discuss your observations.

4.2 NER Experiment

The experiment with the NER data will analyze how changing the domain of the training and testing data can impact the performance of a model.

Instead of accuracy, you will use the F_1 score to evaluate how well a model does. The F_1 score is computed as the harmonic mean of the precision and recall of the classifier. Precision measures how often the model correctly identifies an entity out of how many times the model predicted a word is an entity. Recall measures how many of the actual entities the model

successfully tagged as an entity.

$$\text{Precision} = \frac{\#(\text{Actually Entity \& Model Predicted Entity})}{\#(\text{Model Predicted Entity})} \tag{16}$$

$$\text{Recall} = \frac{\#(\text{Actually Entity \& Model Predicted Entity})}{\#(\text{Actually Entity})} \tag{17}$$

$$\text{F}_1 = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{18}$$

You need to implement the calulation of F_1 yourself using the provided function header. It will be unit tested on Gradescope.

For this experiment, you will only use the averaged basic Perceptron and SVM. Hence, no parameter tuning is necessary. Train both models on the CoNLL training data then compute the F_1 on the development and testing data of both CoNLL and Enron. Note that the model which is used to predict labels for Enron is trained on CoNLL data, not Enron data. Report the F_1 scores in a table.

Comment on the results:

1. Are the F_1 scores on CoNLL and Enron are similar?
2. If they dissimilar, explain why you think the F_1 score increased/decreased on the Enron data.

Rubric

[20 Points] Unit Tests

- [3 points] `calculate_f1`
- [2 points] `Perceptron` implementation (we already did this for you)
- [5 points] `Winnow` implementation
- [10 points] `AdaGrad` implementation

[40 Points] Synthetic Experiment

- [15 points] Averaged Implementations
- [5 points] Parameter Tuning
- [15 points] Learning Curves and answers to questions
- [5 points] Final Test Accuracies
- [+10 points extra credit] Noise experiment

[40 Points] NER Experiment

- [30 points] Feature extraction
- [5 point] Final Test Accuracies
- [5 points] F_1 discussion

Submission Instructions

We will be using Gradescope to turn in both the Python code and writeup pdfs. You should have been automatically added to Gradescope. If you do not have access, please ask the TA staff on Piazza.

For this homework assignment, there are two Gradescope assignments:

– "Homework 2 - Code": This is the assignment where you should upload your implementation of the `calculate_f1` function and the `Perceptron`, `Winnow`, `AdaGrad` classes in a filed caled `hw2.py`. Additionally, you should submit all of your code when you have finished the assignment (upload the individual Python files, zip them together, or upload the Jupyter Notebook as a pdf).

– "Homework 2 - PDF": This is the assignment where you should upload your writeup as a PDF.

Changelog

Version 2

– For the description of the averaged Perceptron, M was corrected to be the total number of correctly classified examples, not the total number of examples.

– Added a clarification on the prediction rule for AdaGrad.