Natural Language Processing

ACADEMIC HONESTY

As usual, the standard honor code and academic honesty policy applies. We will be using automated **plagiarism detection** software to ensure that only original work is given credit. Submissions isomorphic to (1) those that exist anywhere online, (2) those submitted by your classmates, or (3) those submitted by students in prior semesters, will be detected and considered plagiarism.

INSTRUCTIONS

Congratulations on making it to the last programming project. This assignment intends to give you a flavor of a real world AI/ML application, which often requires to gather raw data, do preprocessing, design suitable ML algorithms, and implement the solution. Today, we touch on an active research area in Natural Language Processing (NLP), **sentiment analysis**.

Given the availability of a large volume of online review data (Amazon, IMDB, etc.), sentiment analysis becomes increasingly important. We are going to build a sentiment classifier, i.e., evaluating the polarity of a piece of text being either positive or negative.

The "Large Movie Review Dataset"(*) shall be used for this project. The dataset is compiled from a collection of 50,000 reviews from IMDB on the condition there are no more than 30 reviews per movie. The numbers of positive and negative reviews are equal. Negative reviews have scores less or equal than 4 out of 10 while a positive review have score greater or equal than 7 out of 10. Neutral reviews are not included. The 50,000 reviews are divided evenly into the training and test set.

**Credit: Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. (2011). Learning Word Vectors for Sentiment Analysis. The 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011).*

I. Instructions

Up until now, most of the course projects have been requiring you to implement algorithms discussed in the lectures. This assignment is going to introduce a few advanced concepts of which implementations demand a non-trivial programming expertise. As such, before reinventing the wheel, we would advise you to first explore the incredibly powerful existing Python libraries. The following two are highly recommended: http://scikit-learn.org/stable/ and http://pandas.pydata.org/

Stochastic Gradient Descent Classifier

In this project, we will train a Stochastic Gradient Descent Classifier. Recalled from the Machine Learning project, you were asked to implement a gradient descent update algorithm for linear regression. While gradient descent is powerful, it can be prohibitively expensive when the dataset is extremely large because every single data point needs to be processed.

However, it turns out when the data is large, rather than the entire dataset, SGD algorithm performs just as good with a small random subset of the original data. This is the central idea of Stochastic SGD and particularly handy for the text data since text corpus are often humongous.

You should read sklearn documentation and learn how to use an SGD classifier. For adventurers, you are welcome to manually implement SGD yourself. Wikipedia provides a good description https://en.wikipedia.org/wiki/Stochastic_gradient_descent.

Data Preprocessing

The training data is provided in your starter code. If you wish to download the data to your local machine, use the link: http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz

Your first task is to explore this directory. There are two sub-directories **pos/** for positive texts and **neg/** for negative ones. Use only these two directories. Now **combine the raw database into a single csv files,** "**imdb_tr.csv**". The csv file should have three columns,**"row_number"** and **"text"** and **"polarity"**. The column **"text"** contains review texts from the aclImdb database and the column **"polarity"** consists of sentiment labels, 1 for positive and 0 for negative. An example of "imdb.tr.csv" is provided in the workspace.

In addition, common English stopwords should be removed**.** An English stopwords reference is provided in your starter code for your reference.

Unigram Data Representation

The very first step in solving any NLP problem is finding a way to represent the text data so that machines can understand. A common approach is using a document-term vector where each document is encoded as a discrete vector that counts occurrences of each word in the vocabulary it contains. For example, consider two one-sentence documents:

- d1: "I love Columbia Artificial Intelligence course"
- d2: "Artificial Intelligence is awesome"

The vocabulary V = {artificial, awesome, Columbia, course, I, intelligence, is, love} and two documents can be encoded as v1 and v2 as follow:

	artificial	awesome	Columbia	course	I	intelligence	is	love
v1	1	0	1	1	1	1	0	1
v2	1	1	0	0	0	1	1	0

If you wish to know more, start from here https://en.wikipedia.org/wiki/Document-term_ matrix. This data representation is also called **a unigram model**.

Now, write a python function to transform text column in **imdb_tr.csv** into a term-document matrices using uni-gram model then train a **Stochastic Descend Gradient (SGD) classifier** whose loss="hinge" and penalty="l1" on this data.

In your starter code, you will also find the link to ***"imdb_te.csv.zip"*** which is our benchmark file for the performance of the trained classifier. After unzipping, "***imdb_te.csv"*** has two columns: **"row_number"** and **"text". The** column **"polarity"** is excluded and your job is to use the trained SGD classifier to predict this information. You should transform **imdb_te.csv** using unigram data model as well and use the trained SGD to predict the converted test set. Predictions must be formatted line-by-line and stored in "**unigram.output.txt**". An example of the output file is provided in your starter code.

Bigram Representation

A more sophisticated data representation model is the bigram model where occurrences depend on a sequence of two words rather than an individual one. Taking the same example like before, v1 and v2 are now encoded as follow:

	artifical | artificial	artifical | awesome	···	intelligent | artificial	···	love | love
v1	0	0	···	1	···	0
v2	0	0	···	1	···	0

Instead of enumerating every individual words, bigram counts the number of instance a word following after another one. In both d1 and d2 "intelligence" follows "artificial" so v1(intelligence | artificial) = v2 (intelligence | artificial) = 1. In contrast, "artificial" does not follow "awesome" so v1(artificial | awesome) = v2(artificial | awesome) = 0. Repeat the same exercise from Unigram for the Bigram Model Data Representation and produce the test prediction file "**bigram.output.txt".**

Tf-idf:

Sometimes, a very high word counting may not be meaningful. For example, a common word like "say" may appear 10 times more frequent than a less-common word such as "machine" but it does not mean "say" is 10 times more relevant to our sentiment classifier. To alleviate this issue, we can instead use **term frequency** **tf[t] = 1 + log(f[t,d]**

)where **f[t,d]** is the count of term t in document d. The log function dampens the unwanted influence of common English words.

Inverse document frequency (idf) is a similar concept. To take an example, it is likely that all of our training documents belong to a same category which has specific jargons. For example, Computer Science documents often have words such as computers, CPU, programming and etc., appearing over and over. While they are not common English words, because of the document domain, their occurrences are very high. To rectify, we can adjust using **inverse term frequency** ***idf[t] = log(N / df[t])*** where **df[t]** is the number of documents containing the term t and N is the total number of document in the dataset.

Therefore, instead of just word frequency, tf-idf for each term t can be used, **tf-idf[t] = tf[t] ∗ idf[t].**

Repeat the same exercise as in the Unigram and Bigram data model but apply tf-idf this time to produce test prediction files, "**unigramtfidf.output.txt**" and "**bigramtfidf.output.txt**"

II. What you need to submit:

Your task in this assignment is to write driver.py to produce sentiment predictions over the **imdb_te.csv** by various text data representation (unigram, unigram with tf-idf, bigram and bigram with tf-idf). Please ensure your driver.py write the predictions to the following files **during the run time** (one-time outputs are not accepted):

- unigram.output.txt
- unigramtfidf.output.txt
- bigram.output.txt
- bigramtfidf.output.txt

Please be ***precise*** with these file names because the auto-grader
will rerun your driver.py and look for them for evaluation.

As usual, your program will be run as follows: $python driver.py

If you want to use Python 3 then simply rename driver.py to driver_3.py and your program will be executed as: $python3 driver_3.py