CS440/ECE448 Spring 2022

Assignment 1: Naive Bayes

Due date: Monday January 31st, End of the Day

Updated 2022 By: Xiuye Zhu, Feiyu Zhang, and Mohit Goyal
Updated 2021 By: Ayush Sarkar and Kiran Ramnath
Updated 2020 By: Jaewook Yeom
Updated 2019 By: Kedan Li and Weilin Zhang
Updated 2018 By: Justin Lizama and Medhini Narasimhan

Spam emails are a common issue we all encounter! In this assignment, you will use the Naive Bayes algorithm to train a Spam classifier with a dataset of emails that i
The task in Part 1 is to learn a bag of words (unigram) model that will classify an email as spam or ham (not spam) based on the words it contains. In Part 2, you will c
unigram and bigram models to achieve better performance on email classification.

Contents

General guidelines

Basic instructions are similar to MP 1:

- For general instructions, see the main MP page.
- In addition to the standard python libraries, you should import nltk and numpy.
- Code will be submitted on gradescope.

Problem Statement

You are given a dataset consisting of Spam and Ham (not spam) emails. Using the training set, you will learn a Naive Bayes classifier that will predict the right class la
unseen email. Use the development set to test the accuracy of your learned model, with the included function `grade.py`. We will have multiple hidden test sets that we
your code after you turn it in.

Dataset

The dataset that we have provided to you consists of 1500 ham and 1500 spam emails, a subset of the Enron-Spam dataset provided by Ion Androutsopoulos. This data
2000 training examples and 1000 development examples. This dataset is located in the `spam_data` folder of the template code provided. You will also find another data
the `counter-data` folder of the template code - this data is only used by our grade.py file to check whether your model implementation is correct.

Background

The bag of words model in NLP is a simple unigram model which considers a text to be represented as a bag of independent words. That is, we ignore the position the
and only pay attention to their frequency in the text. Here, each email consists of a group of words. Using Bayes theorem, you need to compute the probability that the
(Y) should be ham (Y=ham) given the words in the email. Thus you need to estimate the posterior probabilities:

$$P(Y=\text{ham}|\text{words}) = \frac{P(Y=\text{ham})}{P(\text{words})} \prod_{x\in\text{words in the e-mail}} P(X=x|Y=\text{ham})$$

$$P(Y=\text{spam}|\text{words}) = \frac{P(Y=\text{spam})}{P(\text{words})} \prod_{x\in\text{words in the e-mail}} P(X=x|Y=\text{spam})$$

You will need to use log of the probabilities to prevent underflow/precision issues; apply log to both sides of the equation. Notice that P(words) is the same in both
can omit it (set term to 1).

Part 1: Unigram Model

Before starting: Make sure you install the nltk package with 'pip install nltk' and/or 'pip3 install nltk', depending on which Python version you plan on using (we sugg
3.8 or 3.9). Otherwise, the provided code will not run. More information about the package can be found here

- **Training Phase:** Use the training set to build a bag of words model using the emails. Note that the method **reader.load_dataset_main** is provided which will al
the training set for you, such that the training set is a list of lists of words (each list of words contains all the words in one email). The purpose of the training set
calculate $P(X=x|Y=\text{ham})$ and $P(X=x|Y=\text{spam})$ during the testing (development) phase.

  Hint: Think about how you could use the training data to help you calculate $P(X=x|Y=\text{ham})$ and $P(X=x|Y=\text{spam})$ during the development (tes
that $P(X=\text{tiger}|Y=\text{ham})$ is the probability that any given word, in a ham e-mail, is the word "tiger." After the training phase, you should be able to com
$P(X=x|Y=\text{ham})$ and $P(X=x|Y=\text{spam})$ for any word. Also, look into using the **Counter** data structure to make things easier for you coding-wis

- **Development Phase:** In the development phase, you will calculate the $P(Y=\text{ham}|\text{words})$ and $P(Y=\text{spam}|\text{words})$ for each email in the development
classify each email in the development set as a ham or spam email depending on which posterior probability is of higher value. You should return a list containin

of the emails in the development set (label order should be the same as the document order in the given development set, so we can grade correctly). Note that y use only the training set to learn the individual probabilities. Do not use the development data or any external sources of information.

Hint: Note that the prior probabilities will already be known (since you will specify the positive prior probability yourself when you run the code) and remembe simply omit $P(\text{words})$ by setting it to 1. Then, your only remaining task is: for each document in the development set, calculate $P(X=x|Y=\text{ham})$ and $P(X=x|Y=\text{spam})$ for each of the words. After that, you will be able to compute $P(Y=\text{ham}|\text{words})$ and $P(Y=\text{spam}|\text{words})$ for each email in set using the formulas above.

Laplace Smoothing: You will need to make sure you smooth the likelihoods to prevent zero probabilities. In order to accomplish this task, use Laplace smoothing. Th Laplace smoothing parameter will come into play. You can use the following formula for Laplace smoothing when calculating the likelihood probabilities:

$$P(X=x|Y=y)=\frac{\text{Count}(X=x|Y=y)+k}{N(Y=y)+k(1+|X|_{Y=y})}$$

where Count(X=x|Y=y) is the number of times that the word was x given that the label was y, k is the Laplace smoothing parameter, $N(Y=y)=\sum_x \text{Count}(X$ the total number of word tokens that exist in e-mails that are labeled Y=y, and $|X|_{Y=y}$ is the number of distinct word types that exist in e-mails labeled Y=y. **A note o vocabulary (OOV) words:** Suppose that the dev set includes any words that never occurred in the training set. In that case, the term $\text{Count}(X=x|Y=y)=0$, terms in the above formula are nonzero, so $P(X=x|Y=y)\neq 0$. This should not be something you compute during your analysis of the training set, but it's a bac you'll need to have available when you analyze the dev set.

Part 2: Unigram and Bigram Models

For Part 2, you will implement the naive Bayes algorithm over a bigram model (as opposed to a unigram model like in Part 1). Then, you will combine the bigram mo unigram model into a mixture model defined with parameter λ:

$$P(Y=\text{Spam}|\text{email})=\left(P(\text{Y}=\text{Spam})\prod_{x_i\in\text{unigrams in the email}}P(X=x_i|Y=\text{Spam})\right)^{1-\lambda}\left(P(Y=\text{Spam})\prod_{b_i\in\text{bigrams in the email}}P(\text{Bigram}_i=b$$

You should not implement the equation above directly -- you should implement its logarithm.

You can choose to find the best parameter λ that gives the highest classification accuracy. There are additional parameters that you can tune (for more details, see here **should note that you can get away with not tuning these parameters at all, since the autograder will choose these values for you when you grade. So feel free with these parameters, but in the end, the hyperparameters you choose when you run on your local machine won't matter in grading.**

Some weird things happen when using the bigram model. One of them happens when you use Laplace smoothing to estimate $P(\text{Bigram}_i=b_i|Y=y)$ from traini Obviously, $\text{Count}(\text{Bigram}_i=b_i|Y=\text{spam})$ is the number of times bigram b_i occurred in spam e-mails, and k is the bigram Laplace smoothing coefficient. Th that $N(Y=\text{spam})$ should equal the total number of bigrams that occurred in spam emails, **plus** the total number of unigrams that occurred in spam emails. Similar should equal the total number of distinct bigrams that occurred in spam emails, **plus** the total number of distinct unigrams that occurred in spam emails. We're not sure model gives higher accuracy with unigrams counted in the denominator, but for some reason, it does.

Part 3: Extra credit

For the extra credit part of this MP, you can try a more informative measure of the importance of each word. **tf-idf** is used in information retrieval and text mining app determine how important a word is to a document within a collection of documents. The tf (term-frequency) term determines how frequently a word appears in a docu kind of related to naive Bayes, though it's not exactly the same thing). The idf (inverse document frequency) term determines how rare (and thus perhaps more inform the collection. A high tf-idf score might indicate that a word has high term frequency in a particular document, and also that it is has a low document frequency (low n documents that contains the given word).

Like a bag-of-words, tf-idf can be used in classifying documents. For the extra credit portion, you will implement tf-idf on the same dataset as Parts 1 and 2. However credit portion is worth only 10%, we will ask you to complete an easier task than in Parts 1 and 2. For this part, instead of classifying the documents, you will find the highest tf-idf value from each of the documents in the development set. Your task is to return a list of these words (with the highest tf-idf values).

For this MP, you should treat the entire training dataset (both ham and spam documents) as the collection of documents from which to calculate your idf term. Your id given word should be calculated based on only the training dataset, not the development dataset. Meanwhile, you should calculate your tf term based on the term frequ document you are evaluating from the development set. Please follow this methodology to obtain results consistent with the autograder.

There are multiple ways in which tf-idf gets implemented in practice. For consistency, we will use the following formula to compute tf-idf for any given word:

$$\text{tf-idf}(\text{word } w, \text{document } A)=\frac{(\#\text{ of times word w appears in doc. A})}{(\text{total }\#\text{ of words in doc. A})}\cdot\log\left(\frac{\text{total }\#\text{ of docs in train set}}{1+\#\text{ of docs in train set containing word w}}\right.$$

If there are terms with the same tf-idf value within the same document, choose the first term for tie-breaking.

Sidenote: A cursory look at the words would show you the importance of governing the outputs of such methods before deploying them on products, but that's beyonc assignment. Interested students could look up literature on bias and fairness in AI.

Provided Code Skeleton

We have provided (tar, zip) all the code to get you started on your MP.

Note that the only files you should need to modify are **naive_bayes.py** for Part 1 and **tf-idf.py** for Part 2. Those are the only files that you will upload to the autograde

Files Used for Part 1&2

- **reader.py** - This file is responsible for reading in the data set. It takes in all of the emails, splits each of those emails into a list of words, stems them if you used flag, and then stores all of the lists of words into a larger list (so that each individual list of words corresponds with a single email). Note that this file is used for assignment (Part 1 and Part 2).
- **mp1.py** - This is the interactive file that starts the program for Part 1, and computes the accuracy, precision, recall, and F1-score using your implementation of n
- **grade.py** - This is a test-grader. It will grade your homework using the provided development test data. The autograder online does some of the same grading, a some hidden tests.

- **naive_bayes.py** - This is the file where you will be doing all of your work for Part 1. The function naiveBayes() takes as input the training data, training labels, smoothing parameter, and positive prior probability. The training data provided is the output from **reader.py**. The training labels is the list of labels correspondi the training data. The development set is the list of emails that you are going to test your implementation on. The smoothing parameter is the laplace smoothing specified with --laplace (it is 1 by default). The positive prior probability is a value between 0 and 1 you specified with --pos_prior. You will have naiveBayes() predicted labels for the development set from your Naive Bayes model.

Do not modify the provided code. You will only have to modify **naive_bayes.py** for Part 1.

mp1.py

Here is an example of how you would run your code for Part 1 from your terminal/shell:

```
python3 mp1.py --training ../data/spam_data/train --development ../data/spam_data/dev --stemming False --lowerc
laplace 1.0 --pos_prior 0.8
```

Here is an example of how you would run your code for Part 2 from your terminal/shell:

```
python3 mp1.py --bigram True --training ../data/spam_data/train --development ../data/spam_data/dev --stemming
lowercase False --bigram_laplace 1.0 --bigram_lambda 0.5 --pos_prior 0.8
```

Note that you can and should change the parameters as necessary. To understand more about how to run the MP for Part 1, run **python3 mp1.py -h** in your terminal.

Optional: How to change your flag

We provide some flags for you to play around when running your code (for help on how to run the code, see here).

For example, you can use the --lower_case flag to cast all words into lower case. You can also tune the laplace smoothing parameter by using the --laplace flag. You ca stemming and --pos_prior parameters. You should be able to boost the model performance up a bit by tuning these parameters. **However, note that when we grade yo use our own flags (stemming, lower_case, pos_prior, laplace).**

grade.py

This file, similar to MP1, is for you to be able to evaluate whether or not you have implemented the unigram and mixture models correctly. This file consists of three f

- It checks if your mixture model approach is correct (whether or not your model is actually a mixture of unigram and bigram models), using the data located und
- It checks if your unigram model is able to reach the correct accuracy threshold for the development set provided
- It checks if your mixture model is able to reach the correct accuracy threshold for the development set provided

Unlike the grade.py file provided to you in MP1, this version provides you with only a subset of the tests that your code will undergo on gradescope. This file should a general idea on whether or not your implementations are correct. Your code will go through more tests on datasets not provided to you once it is submitted to gradesco example of how you would run grade.py from your terminal/shell:

```
python3 grade.py
```

Files Used for Extra Credit

- **reader.py** - This is the same file used in Part 1 and Part 2 (see above for description for this file).
- **run_tf_idf.py** - This is the main file that lets you run the program for the Extra Credit Part.
- **tf_idf.py** - (this is the only file you need to edit) This is the file where you will be doing all of your work for the Extra Credit Part. The function compute_tf_idf the training data, training labels, and development set. The training data provided is the output from **reader.py**. The training labels is the list of labels correspon in the training data. The development set is the list of emails from which you will extract the words with the highest tf-idf values. You should return a list (with s dev_set) that contains words from the development set with the highest tf-idf values. More specifically, the list should contain the word with the highest tf-idf va document in the development set. The order of the words in the list returned should correspond to the order of documents in the training data.
- **grade_tfidf.py** - This file allows you to test your code on the development test provided. We will test your code on unseen test, but testing on dev set should gi sense of whether your implementation is correct or not.

Do not modify the provided code. You will only have to modify **tf_idf.py** for the Extra Credit Part. Here is an example of how you would run your code for the Extra your terminal/shell:

```
python3 run_tests_tfidf.py --training data/spam_data/train --development data/spam_data/dev
```

What to Submit

Submit only your **naive_bayes.py** file to Gradescope, under the assignment called **MP1.**

If you completed the extra credit portion, submit only the **tf_idf.py** file to Gradescope, under the assignment called **MP1: Extra Credit**