CS440/ECE448 Spring 2020

Assignment 3: Naive Bayes

Due date: Wednesday March 4th, 11:59pm

Updated 2018 By: Justin Lizama and Medhini Narasimhan

Updated 2019 By: Kedan Li and Weilin Zhang

Updated 2020 By: Jaewook Yeom

Suppose that we're building an app that recommends movies. We've scraped a large set of reviews off the web, but (for obvious reasons) we would like to recommend only movies with positive reviews. In this assignment, you will use the Naive Bayes algorithm to train a binary sentiment classifier with a dataset of movie reviews. The task in Part 1 is to learn a bag of words (unigram) model that will classify a review as positive or negative based on the words it contains. In Part 2, you will combine the unigram and bigram models to achieve better performance on review classification.

Contents

General guidelines

Basic instructions are similar to MP 2:

- For general instructions, see the main MP page and the course policies.
- In addition to the standard python libraries, you should import nltk and numpy.
- Code will be submitted on gradescope.
- The extra credit portion will be submitted separately on gradescope. In your MP average, the extra credit is worth 10% of the value of the regular part of the assignment.

Extra credit may be awarded for going beyond expectations or completing the suggestions below. Notice, however, that the score for each MP is capped at 110%.

Problem Statement

You are given a dataset consisting of positive and negative reviews. Using the training set, you will learn a Naive Bayes classifier that will predict the right class label given an unseen review. Use the development set to test the accuracy of your learned model. We will have a separate (hidden) test set that we will use to run your code after you turn it in.

Dataset

The dataset consists of 10000 positive and 3000 negative reviews, a subset of the Stanford Movie Review Dataset, which was originally introduced by this paper. We have split this data set for you into 5000 development examples and 8000 training examples. The data set can be downloaded here: zip tar.

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 words appear in, 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 of a review being positive given the words in the review: Thus you need to estimate the posterior probabilities:

$$P(\text{Type} = \text{Positive}|\text{Words}) = \frac{P(\text{Type} = \text{Positive})}{P(\text{Words})} \prod_{\text{All words}} P(\text{Word}|\text{Type} = \text{Positive})$$

$$P(\text{Type} = \text{Negative}|\text{Words}) = \frac{P(\text{Type} = \text{Negative})}{P(\text{Words})} \prod_{\text{All words}} P(\text{Word}|\text{Type} = \text{Negative})$$

You will need to use log of the probabilities to prevent underflow/precision issues; apply log to both sides of the equation. Notice that $P(\text{words})$ is the same in both formulas, so you 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 suggest you use Python 3). Otherwise, the provided code will not run. More information about the package can be found here. You will also have to download the tqdm package in a similar way.

- **Training Phase:** Use the training set to build a bag of words model using the emails. Note that you will already be provided with the labels (positive or negative review) for the training set and the training set is already pre-processed 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 is to help you calculate $P(\text{Word}|\text{Type} = \text{Positive})$ and $P(\text{Word}|\text{Type} = \text{Negative})$ during the testing (development) phase.

  Hint: Think about how you could use the training data to help you calculate $P(\text{Word}|\text{Type} = \text{Positive})$ and $P(\text{Word}|\text{Type} = \text{Negative})$ during the development (testing) phase. Note that $P(\text{Word} = \text{tiger}|\text{Type} = \text{Positive})$ is the probability that a word in a document is "tiger" given that the document is positive. But you can also interpret the likelihood $P(\text{Word} = \text{tiger}|\text{Type} = \text{Positive})$ as being the probability that a document contains the word "tiger" given that the document is positive. Feel free to try both if you would like and see what works better for you (when submitting, we suggest you use the first definition, as it yields better results for the hidden dataset used by the autograder). After the training phase, you should be able to compute $P(\text{Word}|\text{Type} = \text{Positive})$ and $P(\text{Word}|\text{Type} = \text{Negative})$ for any word. Also, look into using the Counter data structure to make things easier for you coding-wise.

- **Development Phase:** In the development phase, you will calculate the $P(\text{Type} = \text{Positive}|\text{Words})$ and $P(\text{Type} = \text{Negative}|\text{Words})$ for each document in the development set. You will classify each document in the development set as a positive or negative review depending on which posterior probability is of higher value. You should return a list containing labels for each of the documents 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 your code should 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 remember that you can simply omit $P(\text{words})$ by setting it to 1. Then, your only remaining task is: for each document in the development set, calculate $P(\text{Word}|\text{Type} = \text{Positive})$ and $P(\text{Word}|\text{Type} = \text{Negative})$ for each of the words. After that, you will be able to compute $P(\text{Type} = \text{Positive}|\text{Words})$ and $P(\text{Type} = \text{Negative}|\text{Words})$ for each document in the development set using the formulas above.

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

$$Likelihood = \frac{count(x) + k}{N + k|X|}$$

where x is a type in X, |X| is the number of types, k is the laplace smoothing parameter, and N is the total number of tokens in X. We believe this equation will become clearer once you write out the posterior probabilities without Laplace smoothing first. In other words, we suggest that you first come up with equations for the posterior probabilities without Laplace smoothing, and then examine the formula above to see how to incorporate Laplace smoothing for your purposes.

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 can also tune the --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 your MP, we will use our own flags (stemming, lower_case, pos_prior, laplace).**

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 model and the unigram model into a mixture model defined with parameter λ:

$$(1 - \lambda)(log(P(Y)) + \Sigma_{i=1}^{n} log(P(w_i|Y))) + \lambda(log(P(Y)) + \Sigma_{i=1}^{m} log(P(b_i|Y)))$$

Please note that you would still have to use log to prevent underflow/precision issues. 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). **However, I 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 to play around with these parameters, but in the end, the hyperparameters you choose when you run on your local machine won't matter in grading.**

Extra Credit

Many of you may have heard of tf-idf, which is short for term frequency-inverse document frequency. tf-idf is used in information retrieval and text mining applications to 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 document, and the idf (inverse document frequency) term determines how many documents in the collection contains a specific word. A high tf-idf score might indicate a high term frequency in a particular document and/or a low document frequency (low number of documents that contains the given word).

Like Naive Bayes, 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, since the extra 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 word with 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 positive and negative reviews) as the collection of documents from which to calculate your idf term. Your idf term for any 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 frequency in the document from the development set. In other words, the tf term should be calculated based on the term frequency in the document you are evaluating from the development set, while the idf term should be calculated based on how many documents in the training set contains the term to be evaluated. This ensures consistency in grading as well.

Use the following formula to compute tf-idf for any given word:

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

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

Provided Code Skeleton

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

Note that the only files you will ever have to modify are **naive_bayes.py** for Part 1, **naive_bayes_mixture.py** for Part 2, and **tf_idf.py** for the Extra Credit part (if you plan on doing the Extra Credit part).

Files Used for Part 1

- **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 the --stemming 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 all three parts of the assignment (Part 1, Part 2, and Extra Credit).

- **mp3.py** - This is the main file that starts the program for Part 1, and computes the accuracy, precision, recall, and F1-score using your implementation of naive Bayes.

- **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, development set, smoothing parameter, and positive prior probability. The training data provided is the output from **reader.py**. The training labels is the list of labels corresponding to each email in 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

parameter you 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() output the 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. Here is an example of how you would run your code for Part 1 from your terminal/shell:

```
python3 mp3.py --training ../MP3_data_zip/train --development
../MP3_data_zip/dev --stemming False --lower_case True --laplace 0.1 --
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 mp3.py -h** in your terminal.

Files Used for Part 2

- **reader.py** - This is the same file used in Part 1 (see above for description for this file).

- **mp3_mixture.py** - This is the main file that starts the program for Part 2, and computes the accuracy, precision, recall, and F1-score using your implementation for Part 2 (mixture of unigram and bigram models).

- **naive_bayes_mixture.py** - This is the file where you will be doing all of your work for Part 2. The function naiveBayesMixture() takes as input the training data, training labels, development set, bigram lambda, unigram smoothing parameter, bigram smoothing parameter, and positive prior probability. The training data provided is the output from **reader.py**. The training labels is the list of labels corresponding to each email in the training data. The development set is the list of emails that you are going to test your implementation on. The lambda and smoothing parameters are all values between 0 and 1 that you specify when you run the program. The positive prior probability is a value between 0 and 1 you specified with --pos_prior. You will have naiveBayesMixture() output the predicted labels for the development set from your new Naive Bayes model.

Do not modify the provided code. You will only have to modify **naive_bayes_mixture.py** for Part 2. Here is an example of how you would run your code for Part 2 from your terminal/shell:

```
python3 mp3_mixture.py --training ../MP3_data_zip/train --development
../MP3_data_zip/dev --stemming False --lower_case True --bigram_lambda=0.5
--unigram_smoothing=0.1 --bigram_smoothing=0.1 --pos_prior 0.6
```

Note that you can and should change the parameters as necessary. To understand more about how to run the MP for Part 2, run **python3 mp3_mixture.py -h** in your terminal.

Files Used for Extra Credit Part

- **reader.py** - This is the same file used in Parts 1 and 2 (see above for description for this file).

- **mp3_tf_idf.py** - This is the main file that starts the program for the Extra Credit Part.

- **tf_idf.py** - This is the file where you will be doing all of your work for the Extra Credit Part. The function compute_tf_idf() takes as input 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 corresponding to each email 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 size equal to that of dev_set) that contains words from the development set with the highest tf-idf values. You should choose the word with the highest tf-idf value from each document in the development set.

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 Credit Part from your terminal/shell:

```
python3 mp3_tf_idf.py --training ../Mp3_data_zip/train --development
../Mp3_data_zip/dev
```

Note that you can and should change the parameters as necessary.

Notes about Submission

We have created three separate MP3 assignments on Gradescope, one for each of the three parts (Part 1, Part 2, and Extra Credit).

Submit only the **naive_bayes.py** file for the Part 1 assignment on Gradescope.

Submit only the **naive_bayes_mixture.py** file for the Part 2 assignment on Gradescope.

If you completed the extra credit portion, submit only the **tf_idf.py** file for the Extra Credit assignment on Gradescope.