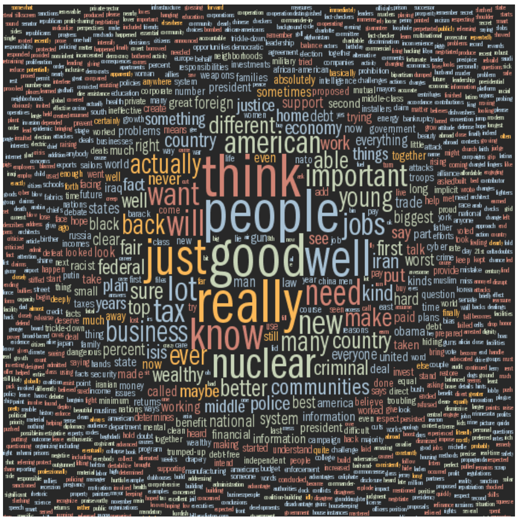

CS440/ECE448 Fall 2021 Assignment 3: Naive Bayes

Due date: Wednesday October 6th, 11:59pm

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 is to learn a bag of words (unigram, bigram) model that will classify a review as positive or negative based on the words it contains.

General guidelines

Basic instructions are the same as in MP 1 and 2. Specifically, you should be using Python 3.8 or 3.9 with pygame installed, and you will be submitting the code to Gradescope. Your code may import modules that are part of the standard python library. You should also import nltk, numpy, and tqdm.

We will load nltk so that we can use its tokenizer and the Porter Stemmer. You may not use any other utilities from the nltk package unless you get permission from the instructor. More information about nltk can be found at the NLTK web site.

The tqdm package provides nice progress displays for the main loops in the algorithm, because they can be somewhat slow when handling this amount of data.

For general instructions, see the main MP page and the syllabus.

Provided code and submission

We have provided a (zip package) containing all the code to get you started on your MP, as well as the training and development datasets.

Submit only your **naive_bayes.py** file on gradescope.

The main program mp3.py is provided to help you test your program. The autograder does not use it. So feel free to modify the default values of tunable parameters (near the end of the file). Do not modify reader.py.

To run the main program, type **python3 mp3.py** in your terminal. This should load the provided datasets and run the naiveBayes function on them. Sadly, it's not doing the required training and just returns the label 0 (i.e. Negative) for all the reviews. Your job is to write real code for naiveBayes and bigram Bayes, returning a list of 0's and 1's.

The main program mp3.py accepts values for a number of tunable parameters. To see the details, type **python3 mp3.py -h** in your terminal. Note that you can and should change the parameters as necessary to achieve good performance.

Dataset

The dataset in your template package consists of 10000 positive and 3000 negative movie reviews. It is 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 autograder also has a hidden set of test examples, generally similar to the development dataset.

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 review 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:

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

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

Notice that P(words) is the same in both formulas, so you can omit it (set term to 1).

Part 1: Unigram Model

Training Phase: Use the training set to build a bag of words model using the reviews. 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 review). 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.

For example $P(\text{Word} = \text{tiger}|\text{Type} = \text{positive})$ is the probability of encountering the word "tiger" in a positive review. After the training phase, you should be able to quickly look up $P(\text{Word}|\text{Type} = \text{positive})$ and $P(\text{Word}|\text{Type} = \text{negative})$ for any word (whether or not it was in your training data).

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 review in the development set. You will classify each review 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 reviews 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.

The prior probability $P(\text{Type} = \text{Positive})$ is provided as an input paramater. You can adjust its value using the command-line options to mp3.py. Inspect the development dataset to determine the actual distribution of reviews in the development data. **Adjust your definition of naiveBayes so that the default value for pos_prior is appropriate for the development dataset.** Our autograder tests will pass in appropriate values for our hidden tests. $P(\text{Type} = \text{Negative})$ can be computed easily from $P(\text{Type} = \text{Positive})$.

Making the details work

Consider Python's Counter data structure.

Use the log of the probabilities to prevent underflow/precision issues. Apply log to both sides of the equation and convert multiplication to addition. Be aware that the standard python math functions are faster than the corresponding numpy functions, when applied to individual numbers.

Zero values in the naive Bayes equations will prevent the classification from working right. Therefore, you must smooth your calculated probabilities so that they are never zero. In order to accomplish this task, use Laplace smoothing. See the lecture notes for details. The Laplace smoothing parameter α is passed as an argument to naiveBayes and you can adjust its value using the command-line arguments to mp3.py.

Tune the values of the Laplace smoothing constant using the command-line arguments to mp3.py. When you are happy with the result on the development set, edit the default values for these parameters in the definition of the function naiveBayes. Some of our tests will use your default settings and some tests will pass in new values.

You can experiment with other methods that might (or might not) improve performance. The command line options will let you transform the input words by converting them all to lowercase and/or running them through the Porter Stemmer. If you wish to turn either of these on for your autograder tests, edit the default values in the function load_data.

You could also try removing stop words from the reviews before you process them. You can add this to load_data or to the start of your naiveBayes function. You will need to find a suitable list of stop words and write a short python function to modify the input data.

No guarantees about what changes will make the accuracy better or worse. You need to figure that out by experimenting.

Bigram Mixture Model

For Part 2, you will implement the function bigramBayes that computes the mixture of unigram and bigram bag of words models. Each bigram b_i is a sequence of two consecutive words from a training or test review. Your bigram code should be very similar to your unigram code, except that you're looking at pairs of words rather than single words. So the probabilities for bigram and unigram models look like this:

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

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

Then you combine the bigram model and the unigram model into a mixture model defined with parameter λ:

$$(1-\lambda)\log\left[P(Y)\prod_{i=1}^{n} P(w_i|Y)\right] + \lambda\log\left[P(Y)\prod_{i=1}^{m} P(b_i|Y)\right]$$

The input to bigramBayes includes two Laplace smoothing parameters, one for the unigram model and one for the bigram model.

The parameter λ controls how much emphasis to give to the unigram model and how much to the bigram model. Choose the value of λ that gives the highest classification accuracy and set this to be the default value of the parameter in bigramBayes.

As with Part 1, some of our autograder tests will use your default values for the tunable parameters inputs to bigramBayes. However, some of our tests will reset these parameter values so that we can test specific aspects of your code. In particular, your definition of bigramBayes should set the default value of pos_prior to match the development dataset and we will adjust this value for our hidden datasets.

You can continue to experiment with stemming, transforming to lowercase, and/or removing stop words. If you turn on these these features from the command line or in load_data, they will be used by both naiveBayes and bigramBayes. You can also use them for only one half of the MP by editing naiveBayes or bigramBayes.