



**Due date: Monday September 11th,
11:59pm**

Page 1 of 4

The main program `mp1.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 mp1.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 -1 for all the reviews. Your job is to write real code for `naiveBayes`, returning a list of 0's (Negative) and 1's (Positive).

The main program `mp1.py` accepts values for a number of tunable parameters. To see the details, type **`python3 mp1.py -h`** in your terminal. Note that you can and should change the parameters as necessary to achieve good performance.

Submitting to Gradescope

Submit this assignment by uploading `naive_bayes.py` to Gradescope. You can upload other files with it, but only `naive_bayes.py` will be retained by the autograder. We strongly encourage you to submit to Gradescope early and often as you will be able to see your final score there.

Policies

You are expected to be familiar with the general policies on the course syllabus (e.g. academic integrity) and on the top-level MP page (e.g. code style). In particular, notice that this is an individual assignment.

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 $P(\text{words})$ is the same in both formulas, so you can omit it (set term to 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 parameter. You can adjust its value using the command-line options to `mpl.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 `mpl.py`.

Tune the values of the Laplace smoothing constant using the command-line arguments to `mpl.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.