Machine Learning **S2018**

Problem Set 5

Handed Out: Mar 26^{th}, 2018 *Due: Apr* 9^{th}, 2018

All the worlds a stage, And all the men and women merely players; They have their exits and their entrances, And one man in his time plays many parts, His acts being seven ages.
–As You Like It, Act II, Scene VII

In this assignment you will use naïve Bayes to classify and generate text! To help you along your way I've provided you with the following files for training: hamlet_train.txt, macbeth_train.txt, romeo_train.txt, juliet_train.txt. I have also provided the following files for testing: hamlet_test.txt, macbeth_test.txt, romeo_test.txt, juliet_test.txt. The training files contain about 80% of the lines spoken by each of these characters (Hamlet, Macbeth, Romeo, and Juliet), while the testing files contain the remaining 20%. These files have been tokenized to make them a little easier to work with. Words have not been stemmed and I have not removed stop words; however, you should be able to split each line of the dataset on spaces to separate each line into tokens. An end-of-line token ($< eol >$) has been added at the end of each line as well.

You will submit a writeup in Word or PDF that summarizes your results and all code as a zip file. Submit the writeup (with attached source code) to the Canvas submission locker before 11:59pm on the due date.

Classification using Naïve Bayes and Unigram Tokens (50 Points)

For the first part of this assignment, use a naïve Bayesian (bag-of-words) approach to classify each of the character lines provided in the test sets. To classify these lines, you will be calculating the probability that each line was spoken to a given character. More formally, you have a class $C = Macbeth, Hamlet, Romeo, Juliet$, and a set of evidence

$E = < word_1, word_2, ..., word_n >$. Your goal is to calculate the following probability for each class: $P(C|E)$. Recall that in class we mentioned that you can use word frequencies and the naïve Bayes assumption to make this task manageable. This should help you on your way in completing this assignment. For the first part of this assignment, only use **unigrams** (individual words) for classification.

In your writeup, please list the overall prediction accuracy for every line, as well as the prediction accuracies for each individual character. So in your writeup, you could have something that looks like this, for example:

- Macbeth Accuracy: XXX%
- Hamlet Accuracy: XXX%
- Romeo Accuracy: XXX%
- Juliet Accuracy: XXX%
- Total Accuracy: XXX%

Details:

- You may not use a high-level function for implementing your approach or performing the classification. You may, however, use math-related functions to make computation a bit easier on yourself.
- All code should be written in Python. You can choose whether you want to use Python 2 or 3, but please specify in your writeup which you use.
- Remember, you are only using **unigrams** for this part of the assignment.
- In the event that you end up with a frequency of 0, use pseudocounts to avoid causing your probabilities to vanish.
- It's probably best to do probability calculations using the log probability to prevent underflow.
- Explain any implementation choices you had to make in the final report.
- Include the accuracy obtained for each individual character, as well as the overall accuracy in your writeup.

Classification using Naïve Bayes and Bigram Tokens (30 Points)

Unigrams are great, but we can do better. Here, extend your previous approach to take advantage of bigram features. A bigram is a sequence of two words or tokens. The sentence

“The quick brown fox” would contain the following bigrams: *The-quick*, *quick-brown*, and *brown-fox*. Bigrams are not limited to words, they can contain other tokens (punctuation or $< eol >$ characters) as well. As before, list out the accuracy obtained for each character as well as on the test set as a whole. How does the accuracy obtained compare to the accuracy values obtained in part 1? Why do you think this is the case?

Details:

- You may not use a high-level function for implementing your approach or performing the classification. You may, however, use math-related functions to make computation a bit easier on yourself.
- All code should be written in Python. You can choose whether you want to use Python 2 or 3, but please specify in your writeup which you use.
- Remember, you are only using **bigrams** for this part of the assignment.
- In the event that you end up with a frequency of 0, use pseudocounts to avoid causing your probabilities to vanish.
- It’s probably best to do probability calculations using the log probability to prevent underflow.
- Explain any implementation choices you had to make as well as the answers to the provided questions in the final report.
- Include the accuracy obtained for each individual character, as well as the overall accuracy in your writeup.

Presentation (20 points)

Your report must be complete and clear. A few key points to remember:

- Complete: the report does not need to be long, but should include everything that was requested.
- Clear: your grammar should be correct and it should be easy to follow what you’re saying.
- Concise: I sometimes print out reports to ease grading, don’t make figures larger than they need to be. Graphics and text should be large enough to get the point across, but not much larger.
- Credit (partial): if you are not able to get something working, explain why in your report. If you don’t explain, I can’t give partial credit.

Bonus: Monologue Generation using a predictive keyboard!

It was once said that a monkey sitting at a typewriter will eventually recreate the works of William Shakespeare if given an infinite amount of time. Let's see if we can do better than that. It turns out that your classifier can very easily be turned into a generator! Why don't we use this to create some new monologues for Hamlet, Macbeth, Romeo, and Juliet.

To do this, instead of using bigram frequencies to diagnose, you can use them to select a sequence of words! So instead of predicting $P(C|E)$, we would predict $P(w_2|w_1)$ where w_1 and w_2 are word1 and word2 respectively. By sampling from this conditional distribution, we can generate brand new text. By doing a random sampling, however, there's a good chance that we'll end up with nonsense. To help make things more coherent, construct a *predictive keyboard*. Instead of sampling from the conditional distribution, you will choose from a subset of the top scoring next words to generate. In other words, you will populate a virtual "keyboard" that you can use to choose the next word to generate.

For an example this, consider the following sentence, "Today I went to the". Using your token frequency model, you choose the following 5 words that are most likely to come next: store, pool, supermarket, airport, bedroom. You would then choose from these words which one should be used to complete the sentence. You can choose as many words as you want to populate your predictive keyboard, but do use at least 5. You are also not restricted to using a bigram model. You may use trigrams or even higher order n-grams if you would like, but do keep in mind that most training sentences are short.

Details:

- As always, don't use any high level functions to extend your code.
- Your predictive keyboard must present at least 5 options to the user to extend sentences.
- Explain any implementation choices you had to make in the final report. This includes things such as bigram/trigram usage.
- If you'd like, you can merge the training and test sets to create one large training corpus. This should result in better monologues.
- Include in your writeup a monologue generated for each character using your predictive keyboard. For this work, a monologue is defined as at least 5 lines of text. Remember, a line must end in the $< eol >$ character.