Xinhao Liao
January 23rd, 2020

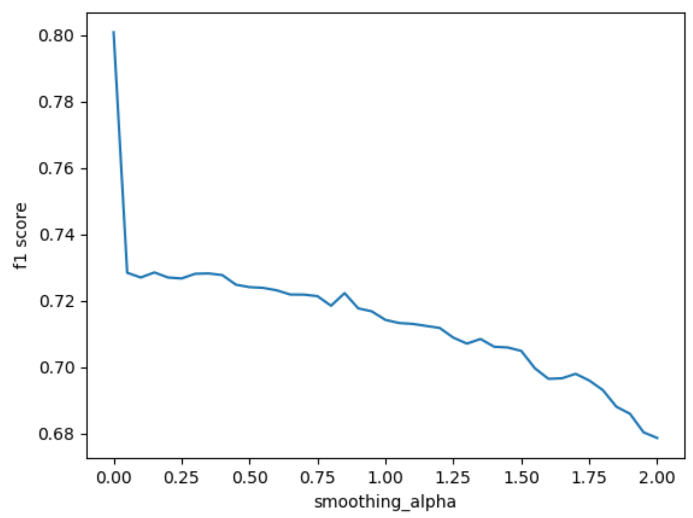
SI630 Homework 1 – Classification

1 Task 1: Naive Bayes Classifier

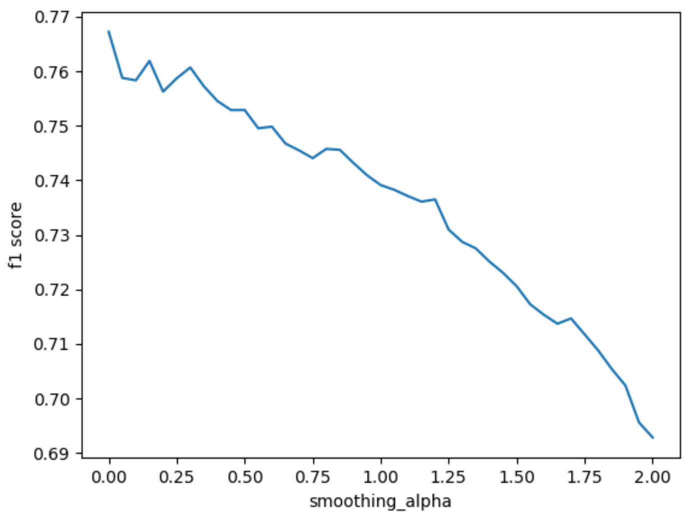
1.1 Part 1

The implementation of required functions is given in *naiveBayes.py*.

In *1-1-smoothing-alpha-vs-performance.py*, the implemented Bayes Network is trained with *smoothing_alpha* values ranging from 0 to 2, and the corresponding alpha values are calculated as shown below. (The two plots correspond to two different implementations of function *classify*, where one ignores unexpected words and the other one not.)



(a) F1 score performance vs. smoothing_alpha, with unexpected words not ignored and original tokenization



(b) F1 score performance vs. smoothing_alpha, with unexpected words not ignored and original tokenization

The best performance occurs when *smoothing-alpha* is set to be 0 and unexpected words not ignored when classifying. So I used this model to generate predictions (codes in *1-1-prediction-result.py*).

1.2 Part 2

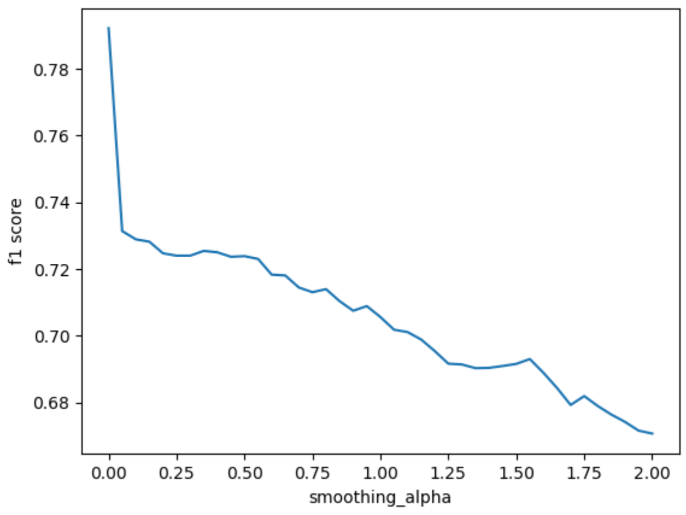
The function *better_tokenize* is implemented in *1-2-better-tokenization.py*. The implementation takes following problems into consideration:

- The case of letters are standardized by making all the letters lowercase with function *lower()*.
- Commas and periods at the end of words are removed using because commas and periods makes no sense in meaning, and "good," should treated the same as "good".
- Words ending with exclamation marks and question marks are split to two parts, the word and the special mark. I choose to keep the marks because exclamation marks and question marks implies some emotional meaning which might be helpful when predicting.

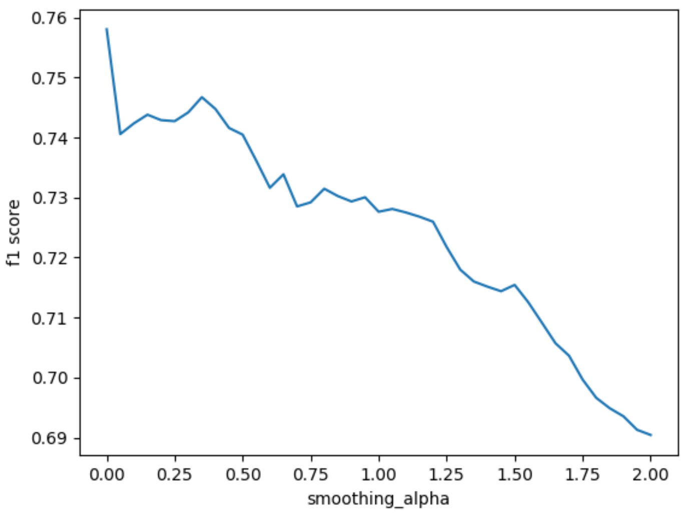
- Commonly used words like "and", "a", and "with" are listed in a list called *stopwords*, and all words in *stopwords* are removed. Such words are used so frequently yet has little sentimental meaning.
- The abbreviations like "'re", "'m", and "'s" are removed because they are actually abbreviations of "am", "is", and "are", and these three words are commonly used words that are not sentimentally meaningful.
- Split words that start with "@" and "http:" are removed because they are used to refer to a user or a web address, and are usually not sentimentally meaningful.

  .

After recomputing the F1-score performance with the *better_tokenization* on development data, the following graphs are obtained. (The two plots correspond to two different implementations of function *classify*, where one ignores unexpected words and the other one does not.)



(a) F1 score performance vs. smoothing_alpha, with unexpected words not ignored and better tokenization



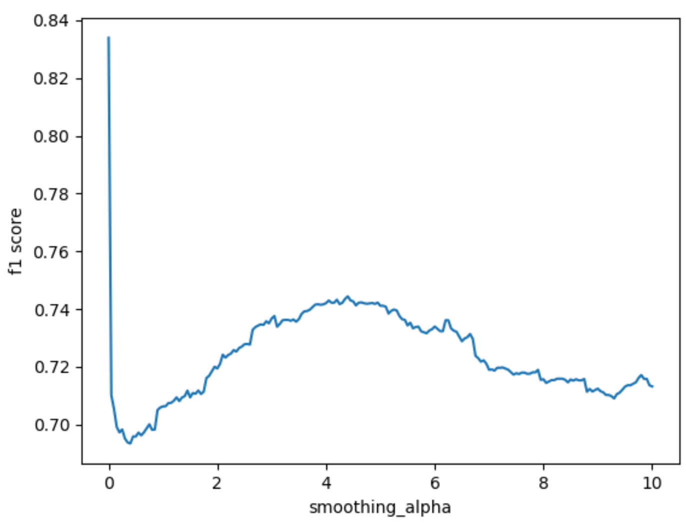
(b) F1 score performance vs. smoothing_alpha, with unexpected words not ignored and better tokenization

As we can see, *better_tokenization* generally generate results with worse performance. (Even when I try to remove some of the implementations listed above, the performances are generally not better than the original ones.)

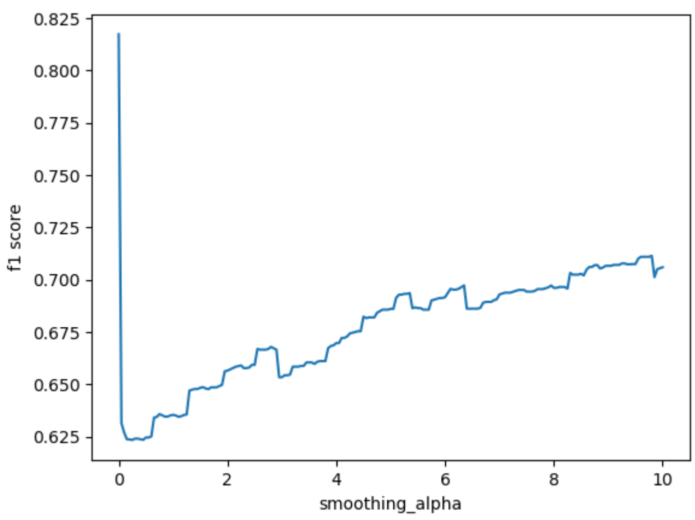
1.3 Part 3

We can further train the model with bigram and trigrams. It is found that there are 21429 unique unigrams, 66962 unique bigrams, and 85597 unique trigrams in the training data. For the worst case, 21429 unique unigrams can at most have $21429^2 = 459202041$ unique bigrams, and $21429^3 = 9.84 \cdot 10^{12}$ unique trigrams. The bigram and trigram counts I can observe are far fewer than the worst case. It's natural because words are not equally common. According to Zipf's law, "the frequency of a word is inversely proportional to its rank in the frequency table" [1]. So we tend to see far fewer word pairs than that could exist.

By changing smoothing_alpha, we can obtain the following graphs showing the F1 score performances. As we can see, we can achieve best performance using bigram data. And prediction results are also generated with this model.



(a) F1 score performance vs. smoothing_alpha for bigram data, with unexpected words not ignored



(b) F1 score performance vs. smoothing_alpha for trigram data, with unexpected words not ignored

For bigram data, some tokenization techniques like adding "¡start¿" and "¡end¿" tokens, separating marks and words can also slightly improve the performance. The code is in *1-3-n-grams.py*.

2 Task 2: Logistic Regression

The implementation of functions are given in *logisticRegression.py*. The model is trained with learning rate of 0.05, 5e-5, and 5e-8 for 1000 step, and the corresponding log-likelihoods are calculated and plotted as below. (The code is given in *2-1000_steps_graph.py*.)

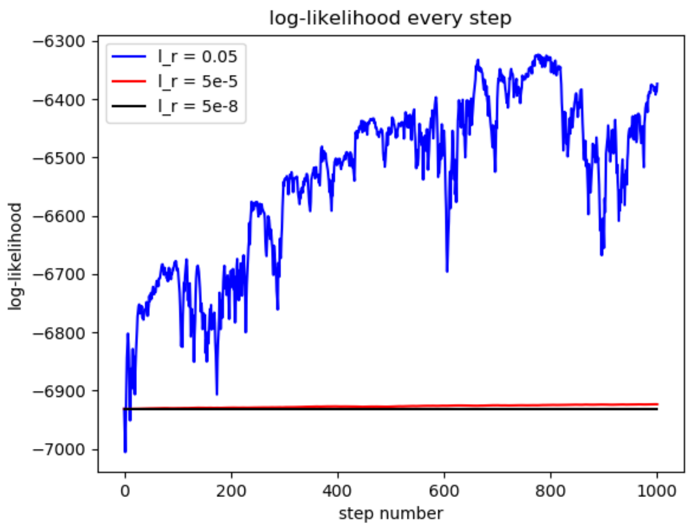


Figure 4: log-likehood every step for learning-rate = 0.05, 5e-5, and 5e-8.

As we can see, none of the models converge within 1000 steps, yet the log-likelihood of models with larger learning rates grows faster.

We can further adjust the parameters to make the curve converge. Finally, I set learning rate = 0.25 and numstep = 400000, and obtain the following graph.

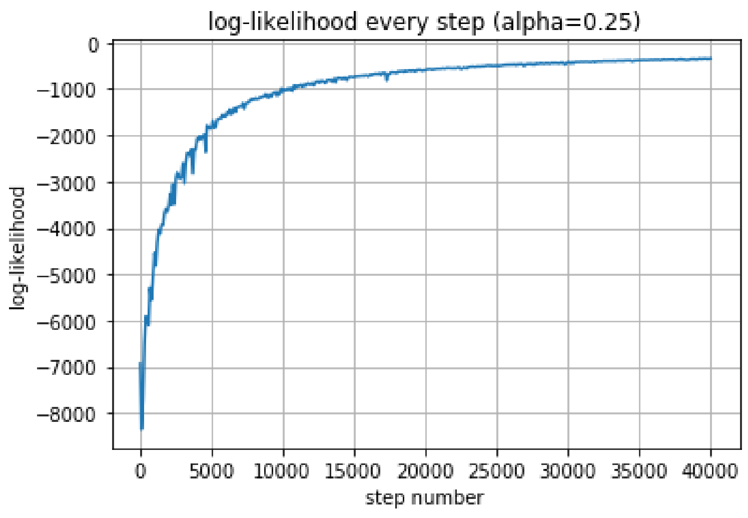


Figure 5: log-likehood every step for learning-rate = 0.25 and numstep=400000.

This converged result gives the F1 score performance of about 0.8026. Further, we can generate prediction with the training result of B. All the code is given in *2-400000_steps_graph_and_result.py*.

Using the bigram model which gives the best result in Task 1, we can train the model and compute the log-likelihood as follows. As we can see, the log-likelihood grows faster than before. And the f1 score performance is around 0.8082, which is slightly higher than before. The generated the prediction result from this model also has the best performance. Code is given in *2-bigram_result.py*.

Note: The code was initially run in colab, and sparse matrix is not implemented. So running on local machines might lead to memory crashes.

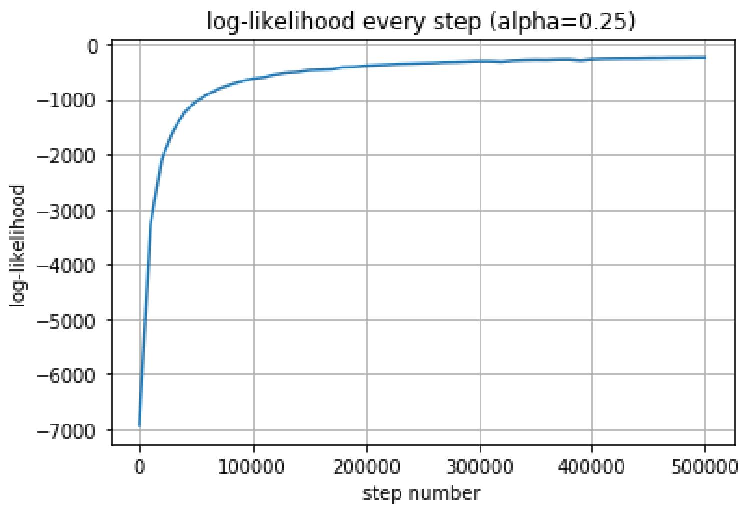


Figure 6: log-likehood every step for learning-rate = 0.25 and numstep=500000 with the model fed with bigram data.

References

[1] "Zipf's law", *Wikipedia, the free encyclopedia.* `https://en.wikipedia.org/wiki/Zipf%27s_law`.