**Part 4) Smoothing and unknown words**

The goal was to implement smoothing and a method for handling unknown words.

For smoothing, we decided to implement +k smoothing. We debated trying +1 smoothing, but because +1 smoothing is generally inadequate, we jumped straight to +k. To find the k values, we implemented a two stage process, where we would first increase k values by 10 (from k=1\*10^-9 up to k=1), then build a new range around the best k value to narrow it down further.

To evaluate the effectiveness of a k value, we built an updated probability model for them with the training data by summing the count numerator by k and denominator by k\*N, where N is the number of tokens. We then evaluated the effectiveness of this model on the validation data set, for both positive and negative data sets. We evaluated the effectiveness by adding the negative log probabilities for the validation corpus, and then choosing the k that minimizes the perplexity of the dataset.

The results from this process will be discussed in Part 5), as that is where we calculated the perplexity.

For unknown words, we decided to assign the count for all words that occurred once and only once to the unknown counter, *<unk>*. This was a standard method discussed in class that we felt would perform sufficiently well. We then calculated the probability distribution model with these updated counts. For unigrams, this was sufficient to include unknown words in the probability mass. We also kept track of which words we'd converted into unknown words. Then for bigrams, if either of the two tokens in the bigrams were in the unknown words set, we replace it (for the purpose of the count, not in the actual data set), and increment the resulting (*<unk>*, token), (token, *<unk>*), (*<unk>*, *<unk>*) entry accordingly. After this, we then convert the counts into a suitable probability distribution, with +k smoothing as described above.

For unseen bigrams, we had also added a 0 count probability for every unseen combination of word types. This was incredibly inefficient, and unnecessary. Any bigram in the validation/test data set that had both its tokens in the training corpus, but not the combination together, could be assumed to be an unseen bigram, for which would assign a default probability, non-zero because of our +k smoothing. We did not apply unseen ngrams to the unigram, as we couldn't see how we would ever have unseen unigrams. Any unigram that appears in the training corpora will either have a count of 1, and be treated as unknown, or more than 1, and have a count assigned. This is different for bigrams, because we could have the possibility where the individual tokens all appeared in the training corpus, just not in the specific combination.

**Part 5) Perplexity**

Here, the goal was to implement a mechanism for evaluating the perplexity of our models. We used the formula provided within the project description, with no major modification. Note that the formula described the sums of the conditional probabilities of a token given the rest of the words in that n-gram. This is essentially what we calculated as our language model in Part 3, and enhanced with smoothing in Part 4. With Part 5, we were now able to evaluate the effectiveness of our probability models, smoothed with different values k. The results are displayed below:

|  |  |  |
| --- | --- | --- |
| Test case | Perplexity | K value |
| Positive Unigram | 1.78 | 1 \* 10^-9 |
| Positive Bigram | 2.01 | 0.011 |
| Negative Unigram | 1.76 | 1 \* 10^-9 |
| Negative Bigram | 1.97 | 0.011 |

To test perplexity, I ran through a logarithmic scale from 1\*10^-9 to 1, increasing by \*10 every time. Then, with the smallest perplexity value k, I went through a smaller scale centered around k, to try and finetune the optimal k value.

As we see for unigrams, the k value is minimal. This likely indicates that we are overfitting on the data for unigrams, and that our eventual model will not handle unseen tokens well (tokens it has seen before will have a disproportionate effect on the end result). This is likely because unigrams do not have enough context, and so we are shifting over too much probability mass for no reason. While we left it as is to see what impact it would have, we would normally add some nominal k value anyways to avoid overfitting our model.

For bigrams, we observe different results. The optimal k value, when the perplexity of the model is minimized, is at 0.011. This is a good result, as it illustrates the concave curve we expected to see in our dataset. I separated both the k for positive and negative models, even though they ended up being the same. This shows that we shifted over neither too much nor too little probability mass. While we may still be overfitting on our models, it’s less likely. I then used this computed k value for Part 6, when I used language models to classify sentences based on their sentiment.

A further enhancement we could have made on this would have been to stagger how we added/removed probability mass. That is, instead of just adding +k across the board, we could have added more probability mass to tokens that showed up rarely, and removed it from tokens that appeared more frequently in the training set.

**Part 6) Sentiment Classification**

Here, the goal was to perform sentiment classification. When we saw this, our immediate thought process was that in previous parts, we'd already built separate probability models for positive and negative corpora, and that we could leverage those here. We decided to adopt a similar approach to how we'd handled perplexity.

We first split up up the (validation and test) corpora into sentences, and preprocessed as necessary. We'd decided to test it with unigrams first, so we simply fed each token into our probability model, then summed up the negative log of the ensuing product. We then compared the outputs from positive and negative probability models, and classified the sentence based on which had the higher resulting probability. We then repeated the same process with bigrams instead of unigrams.

We then submitted two runs to Kaggle. The first was a unigram model with +1 smoothing, the simplest test case, just to make sure that our process was working. The second was the bigram model with +k smoothing we had developed. To our surprise, it had a worse performance than unigram model with +1 smoothing. We will discuss why after reviewing the outcome from our validation data set.

Results from Kaggle:

|  |  |
| --- | --- |
| Run Type |  |
| Unigram model with +1 smoothing | 0.58043 |
| Bigram model with +k smoothing | 0.55924 |

We also tested against our validation data sets. The results are below.

When we treat every 1 count word as unk:

Expected all sum\_pos\_uni, but got 299 sum\_pos\_uni and 118 sum\_neg\_uni

Expected all sum\_pos\_bi, but got 277 sum\_pos\_bi and 140 sum\_neg\_bi

Expected all sum\_neg\_uni, but got 206 sum\_pos\_uni and 202 sum\_neg\_uni

Expected all sum\_neg\_bi, but got 207 sum\_pos\_bi and 201 sum\_neg\_bi

When we only treat every second 1 count word as unk:

Expected all sum\_pos\_uni, but got 285 sum\_pos\_uni and 132 sum\_neg\_uni

Expected all sum\_pos\_bi, but got 267 sum\_pos\_bi and 150 sum\_neg\_bi

Expected all sum\_neg\_uni, but got 240 sum\_pos\_uni and 168 sum\_neg\_uni

Expected all sum\_neg\_bi, but got 217 sum\_pos\_bi and 191 sum\_neg\_bi

When we only treat every third 1 count word as unk:

Expected all sum\_pos\_uni, but got 269 sum\_pos\_uni and 148 sum\_neg\_uni

Expected all sum\_pos\_bi, but got 258 sum\_pos\_bi and 159 sum\_neg\_bi

Expected all sum\_neg\_uni, but got 214 sum\_pos\_uni and 194 sum\_neg\_uni

Expected all sum\_neg\_bi, but got 203 sum\_pos\_bi and 205 sum\_neg\_bi

When we treat every fourth 1 count word as unk:

Expected all sum\_pos\_uni, but got 258 sum\_pos\_uni and 159 sum\_neg\_uni

Expected all sum\_pos\_bi, but got 245 sum\_pos\_bi and 172 sum\_neg\_bi

Expected all sum\_neg\_uni, but got 227 sum\_pos\_uni and 181 sum\_neg\_uni

Expected all sum\_neg\_bi, but got 203 sum\_pos\_bi and 205 sum\_neg\_bi

Looking at our results, we observed some very strange trends. For positive test inputs, our unigram model performed better than our bigram model. For the negative model, it performed worse than the bigram, and even than a coin toss. For the Kaggle data, we couldn’t distinguish positive and negative input, but we could see that the unigram model with +1 smoothing, our initial test point, was marginally better than bigram model with +k smoothing. At this point, we evaluated how we were assigning counts to unknown, and decided to reduce the number of unknown words by only assigning every second, third or fourth 1 count token to be unknown. This did not help the performance of the system measurably, so we kept using every unknown word

When looking at why this was the case, we came up with a few theories. First, our data set is very small. This could cause outliers in the training data to have an adverse effect on our model. In addition, we did not remove filler words as part of our preprocessing. At the time, we though it would not be necessary, but now, looking at the results, we suspect it had an adverse effect.

When we referenced literature to find what a "good" sentimentality result would have been, Roebuck (2012) showed that a program with 70% accuracy is actually close to how a human performs. This indicates that our positive unigram probability model, where we treat every 1 count token as *<unk>* is actually performing very well, and our negative probability model, for both unigram and bigram and across all test cases is not. We could not identify a specific reason, other than the dataset itself, for why the performance was so poor. A future improvement could be to also expand from bigrams to trigrams or quadgrams, in the hope that this extra context would capture more accurately the sentiment of the statement.

Part 7)

**Part 8.1) A machine learning variant of sentiment classification with word embedding**

We decided to go with the Machine Learning approach and chose part 8.1. The high level approach that we followed for this task was to:

* Come up with a good feature representation of the movie reviews in the training data
* Train a Machine Learning model that gives good accuracy on the ‘dev’ dataset
* Use the ‘dev’ dataset again to perform cross-validation to tune the hyper parameters of our model
* Finally make predictions using the tuned model on the test data.

We tried a couple of ML algorithms that we were familiar with and that we expected to do well for the binary classification task we were given. We decided to start with linear classifiers, as they are not only simple, but also well suited for the initial problem, and we were expecting an acceptable accuracy.

We therefore first tried with Naïve Bayes and simple SVMs. \*Talk about Accuracy\* Having found that these classifiers were not actually giving us a good accuracy, we decided to switch to non-linear models by using kernels. We knew that Radial Basis Function Kernel (RBF Kernel) are a universal approximator and work well in most scenarios. After playing around a bit with the “C” value which determines the amount of slack for allowing points within the margins of the SVM, we were able to train a classifier that gives us 81.28%. We determined the ideal value for “C” using cross-validation on the “dev” dataset on which we were getting 72.48 % accuracy(max). We used K-fold Cross Validation with 10 folds (K=10). While more splits could have been possible, it would have led to more iterations and in-turn significantly higher validation time.

For coming up with a good feature representation of the movie reviews, we first used the Word2Vec functionality in Python from the “genism” library to create the word embeddings ourselves from the given corpus. But this was giving us very bad accuracy, as neither SVM nor Naïve Bayes was able to learn a good enough decision boundary. After looking at the discussions on Piazza, we realized our mistake on the way we were generating features for the model, and decided to experiment with some of the pre-trained word embedding suggested in the problem document. We used the pre-trained word vectors from Google News dataset which has 300-dimensional vectors for 3 million words and phrases. Since this significantly improved the accuracy, we did not experiment with the other pre-trained embeddings like Glove.

References:

Roebuck, K. (2012-10-24). Sentiment Analysis: High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors