CS 4740/5740 – Project 1 Report

Ishaan Jain(irj4)

Darpan Kalra(dk557)

Tommy Shum(ts539)

Notes:

- All of our code was written in Python

- We used the following external libraries in our code: numpy, Word2vec

Part 2:

In part 2, we wrote a program to calculate unsmoothed unigrams and bigrams probabilities from a corpus. We decided to count punctuation marks as a word type and word token. However, during preprocessing, we removed punctuation from the beginning and the end of sentences. This is because punctuation does not make sense at the beginning of a sentence, and we replace the punctuation at the end of the sentence with a </s> token. For each sentence, we added a start token <s> to the beginning and an end token </s> to the end. We decided to count uppercase and lowercase words as different word types. We decided to do this because uppercasing can denote that the word belongs at the beginning of the sentence or that it is a proper noun.

Our program makes a single pass through the corpus and keeps count of unigram and bigram instances in a dictionary. Then we calculated the probabilities for each unigram by dividing the count for each unigram by the total word count. To calculate bigram probabilities, we divided the bigram counts by the unigram count of the preceding word.

Part 3:

For part 3, we developed a random sentence generator using our unigram and bigram language models. To generate random unigram sentences, we start with an empty list. Then we choose random tokens from the available unigram tokens and add it to the current sentence. The tokens are chosen with probability equal to the unigram probability computed for this token in part 2. We did this so that tokens would be chosen with probability proportional to their unigram probabilities.

To generate random bigram sentences, we start with a start token <s>. We do this in order to choose words that make sense at the beginning of a sentence. Then we take the last token in the sentence, and choose a random bigram that includes that token with probability equal to the

bigram probability calculated in part 2. We stop when we add an end token </s> to the sentence. Again, we do this so that we end our sentences with appropriate words.

Next, we experimented with seeding, where we started with incomplete sentences, and completed the sentences using our language model. We did not do seeding for unigrams, as this does not make sense. This is because unigram sentences do not depend on any previous words in the sentence. For the bigram sentences generated with seeding, we started with three arbitrary starting phrases:

- "The movie was…"
- "I am…"
- "The film…"

We looked through the text files to get an idea of what starting phrases were common. We decided on these starting phrases, because they are broad and general and can be completed in many different ways. Then the random sentence generator takes the last word in starting phrase and picks a bigram phrase based on that word and adds it to the sentence. Then our random sentence generator proceeds as usual for bigram sentences and continues until we add an end token </s> to the sentence.

Below are some sample sentences generated by our language model

Positive Unigram Sentences:
of about writer\/director offering as is and trust years Bound

Negative Unigram Sentences:
drudgery In the genuinely the The , writer-director , the

Positive Bigram Sentences:
<s> A big-budget\/all-star movie that keep the line , like those </s>

Negative Bigram Sentences:
<s> A culture clash between not-very-funny comedy that was the duration </s>

Positive Bigram Sentences with seeding:
The movie was so that freshly considers arguments the film pantheon </s>

Negative Bigram Sentences with seeding:
The movie was a relatively short of wreckage that , it also looking </s>

Part 3 - Analysis:

The unigram sentences were generally unintelligible and did not convey negative or positive sentiments effectively. The bigram sentences were generally better quality and made more sense. For example, the bigram sentences always correctly start with a capitalized word, but this

not true of the unigram sentences. Also, the unigram sentences sometimes incorrectly contain capitalized words in the middle of the sentence. However, this problem never occurs in sentences generated by the bigram model. The bigram sentences generated by seeding were of similar quality to the bigram sentences generated from scratch. This makes sense, since bigrams are chosen based on the preceding word. Therefore, the bigram sentences generated by seeding chose phrases that made sense in the context of the proceeding word. We did not do seeding for unigram sentences, since it does not make sense.

Part 4:

For part 4, we implemented smoothing and a method to handle unknown words in the test data. The method we used is add-k smoothing. In order to find k, we took dev dataset, and we took the summation of the negative log bigram probabilities. We chose the k value that minimized this summation.

To handle unknown words, we replaced all words with a count of one in the dev dataset with unknown <UNK>.

To handle unseen bigrams in the form <UNK> | i, we smooth this using the formula below

$$\frac{k}{unigram\ count(i) + kV}$$

Where k is the optimal k value we found earlier and V is the size of the vocabulary. We decided to smooth bigrams because otherwise unseen bigrams will have a probability of 0. Therefore, the probability of forming a sentence with this bigram is 0.

We did not do smoothing for unseen unigrams because this does not make sense. Unseen unigrams are already accounted for with <UNK>.

Part 4 - Analysis:

k_value: 1e-05 curprob: 59212.35596230844
k_value: 0.0001 curprob: 53127.04755394552
k_value: 0.001 curprob: 48058.55253862514
k_value: 0.002 curprob: 47018.36260222526
k_value: 0.003 curprob: 46553.43655023914
k_value: 0.004 curprob: 46292.405202380665
k_value: 0.005 curprob: 46130.533437071215
k_value: 0.006 curprob: 46025.073151857985
k_value: 0.007 curprob: 45954.91827270233
k_value: 0.008 curprob: 45908.33004873625
k_value: 0.009 curprob: 45878.21958724567
k_value: 0.01 curprob: 45860.03895486515
k_value: 0.011 curprob: 45850.731690844834
k_value: 0.012 curprob: 45848.16671334252

k_value: 0.013 curprob: 45850.8131192668
k_value: 0.014 curprob: 45857.5436540485
k_value: 0.015 curprob: 45867.51087852545

We tested k values ranging from 0 to 1, and found that k = .012 is optimal. We did not test k values greater than 1, because that would mean shifting too much massing.

Part 5:

For part 5, we implemented a method to compute perplexity of the unigram model and the bigram model.

Part 5 - Analysis:

Positive Bigram Perplexity: 1.1641641528415458
Positive Unigram Perplexity: 1.1625265595212373

Negative Bigram Perplexity: 1.1581514508120327
Negative Unigram Perplexity: 1.15941183526359

Our unigram model had a lower computed perplexity as opposed to our bigram model. This means that our bigram model is more complex. This was to be expected since the bigram model uses two words as opposed the unigram model which uses only one word.

Part 6:

For part 6, we developed a method to use language models to predict sentiment of unlabeled movie reviews. We first computed the probability that a sentence is positive by using the unigram and bigram probabilities obtained from the positive training corpora. When using unigrams, we looked at each word in the unlabeled review and retrieved the positive unigram probability for that word. Then we multiplied all the probabilities together to determine the probability that this unlabeled movie review is positive based on our positive unigram model. Similarly, when using the bigram model, we retrieve the positive bigram probability for each pair of words in the unlabeled review and multiply them together. Next, we computed the probability that a sentence is negative by using the unigram and bigram probabilities obtained from the negative training corpora. Then we compare the two probabilities to see which is higher and we chose the sentiment with higher probability.

Part 6 - Analysis:

According to our test results, the bigram model was better than the unigram model at predicting sentiment of unlabeled movie reviews.

Below are the test results for the unigram and bigram model:

Bigram:
Accuracy: 0.522424242424
Classified: 431 out of 825 samples correctly

Unigram:
Accuracy: 0.352727272727
Classified: 291 out of 825 samples correctly

Part 8.1:

For part 8.1, we trained a classifier to predict the sentiment of unlabeled movie reviews. First, we used Word2vec to compute the word vectors for each training corpus and test corpus. We decided to use Word2vec, because it is openly available, easy to understand and efficient. We chose to compute word vectors for the test corpus, so that we do not have unseen words. After computing the word vectors, we used the vectors to train classifiers.

We first started with a Kernelized Support Vector Machine. We thought that Kernelized SVM would be effective since it is good at learning similarity between features, which seemed to fit our task well. However, the Kaggle results for our Kernelized SVM classifier were worse than expected. So afterwards, we experimented with Random Forest and Gradient Boosted Classifier.

Part 8.1 - Analysis:

Below are the Kaggle results for each of our classifiers:

Random Forest:
Accuracy: 0.670303030303
Classified: 553 out of 825 samples correctly

Gradient Boosted Classifier:
Accuracy: 0.780606060606
Classified: 664 out of 825 samples correctly

Kernelized SVM with c = 100:
Accuracy: 0.810909090909
Classified: 669 out of 825 samples correctly

Kernelized SVM with c = .01:
Accuracy: 0.505454545454
Classified: 417 out of 825 samples correctly

Gaussian Bayesian Classifier:
Accuracy: 0.780606060606
Classified: 664 out of 825 samples correctly

Linear SVM:
Accuracy: 0.796363636364
Classified: 657 out of 825 samples correctly

Workflow:

For part 2, Tommy wrote the code for preprocessing the sentences in the corpus. Ishaan wrote the code for calculating unsmoothed unigram and bigram probabilities.

For part 3, Darpan wrote the code for generating random sentences with unigram and bigram models. Darpan also wrote the code for generating bigram sentences with seeding.

For part 4, Darpan wrote the code for k-smoothing.

For part 5, Darpan wrote the code for calculating perplexity of the test set.

For part 6, Ishaan and Tommy wrote the code for using language models to perform sentiment classification. Ishaan was responsible for submitting the Kaggle predictions.

For part 8.1, Ishaan wrote the code to use Word2vec to generate vectors. Ishaan also wrote the code for training a classifier based on these vectors. Ishaan also submitted the Kaggle predictions for part 8.1.

Tommy wrote the Project 1 Part 1 Analysis. All three members contributed to the final Project 1 Report.