CS 5740 Project 1 Report

Haotian Pan (hp343), Jiang Jian (jj544), Jue Zhang (jz578)

In this project, we designed a language model, which consists of the following components as elaborated in subsequent sections.

PART 0 Pre-processing
For the pre-processing procedure, in order to analyze the corpus easier and better, we decided to separate every sentence with "<s>" and "</s>" marks, for <s> at the beginning of the sentence and </s> at the end. This seemingly trivial trick has some rationale behind it. This could model the probability of a word to be at the beginning or the end of the sentence, which can be exploited by various subsequent tasks such random sentence generation, as shown in the following sections.

Also we handled the punctuation and special expressions such as "'t" or "'d" during tokenization. As the sentences in the Bible corpus are marked for every chapter with number,, we first deleted all those numbers. Then we tried to find the end of the sentence and marked them with "<s>" and "</s>".

We used Splitta to pre-process the hotel corpus the first time. However, the results did not seem to meet our requirements because it separates all the sentences instead of keeping each review together and marking each sentence. So we developed our own Python program based on some rules from Splitta. While the hotel reviews are not that well organized as the Bible corpus, we first deleted all the metadata in every corpus. We developed rules to keep each review together and add the "<s>" and "</s>" marks to separate each sentence. We know there would be some problems so we developed additional codes to solve the unexpected tokens, like "Mr. </s>". For the reviews of training and validating of hotel corpus, we kept the headers, like "0,0," of each review for further analysis . But for the reviews of testing of hotel corpus, we deleted all headers since it means little for us.

For the Kaggle test corpus, it is similar processing strategy as the testing reviews of the hotel corpus.

PART 1 Unsmoothed ngrams
The unsmoothed n-gram model is encapsulated in a class named *LangModel*, which consists of two main functionalities, train and test.

When an object of *LangModel* is created, the constructor takes the corpus as input, trains the language model, stores the trained model into files for future usage as well as load the model to several fields of the class to facilitate the test task.

For training, the core data structure leveraged is a Dictionary (in Python), which is handy to store the counts of n-grams where the keys correspond to the n-gram, and the values are the actual counts. A linear scan on the corpus suffice to collect all the counts we need, which are then dumped to a file (using Pickle, a library for serializing and de-serializing a Python object structure) to store the trained model.

Then we provide a load_model() method that can load the stored model from model file to the memory to compute probabilities on testing.

Finally, for testing, two methods *prob_Unigram()* and *prob_Bigram()* are provided to compute the unsmoothed probabilities for unigram and bigram models respectively.
For unigram model:

$$Prob(w) = \frac{count(w)}{N}$$

where N is the number of total tokens.
For bigram model:

$$Prob(w_1|w_2) = \frac{count(w2,w1)}{count(w_2)}$$

where $count(w_2, w_1)$ is the number of occurrences of bigram (w_2, w_1) .

PART 2 Random sentence generation
In this part, we generated random sentences based on unigram, bigram and trigram language models trained from both the bible corpus and the hotel corpus. Before submission of project1 part1, these models had not been smoothed yet, we now present results based on good-turing models.

These are the sample results of the generation, three Sample sets for each corpus:

	From Bible corpus	From Hotel Corpus
Unigram Model	(1) The there before shall chariot their people they us Go the called the upon , the And father there , defiled , , that swear (2) Emins more heads . him . the oxen the city brethren of . little , the king to my their He wilderness certain earth upon (3) stone thing were and with least I thee shall the didst , the kindle the And thousand glory had the Egypt are of came and	(1) they rock to small night to room. not when now very no then hotel just a forward to insane top of on ease for again (2) good great I banker family if hotel walls professional. up large most hotel know Park but customer talking to carpeting Advisor no less frustratingly spotless (3) again. - reason worth with even a with 3 How hotel. need up Holiday the ! me extensive they word downtown hotel " look and

Bigram Model	(1) And if it came forth his servants , and the latter husband arose , and the water . (2) And Joshua and he found in the LORD shall be blessed him : for thou stoodest in Etham , Speak , which lieth shall not (3) And in the name Enos : for evil was brought that were defiled the word was established .	(1) booked my clients. (2) daughter and the bathroom was given a great selection of time I come back. (3) room was somewhat reluctant to find anything you first floor and the rooms on the hotel ; I read just was arranged so first arrived
Trigram Modell	(1) And he made curtains of the earth , in silver , and will judge his people : anc there shall be deaf . (2) The rich man had nothing , then shall he return . (3) And LORD had respect unto Abel and to pour out my net upon them .	(1) We stayed at the Affinia Chicago is a beautiful room with double sinks in the morning . (2) At checkout ; and not get it together ; and how can you believe it . (3) We didn 't test out the extra money for a business trip to Chicago .

The main problem for random sentence generation is to pick up words from the corpus at exactly the same probabilities that they appear in the training corpus. Our approach is to restore the corpus from the language model we built in Part 1 by adding exactly what is counted in modeling into a pool. The reason why we chose to do this instead of deriving choices from probabilities calculated in the model is that by retrieving corpus from the dictionary in the model, we could represent the pool of words losslessly.

We also make use of the special tokens '<s>' and '</s>' in random sentence generation. In unigram models, we did not include the token '<s>' into the pool, because we do not think it is meaningful to insert an '<s>' which represents the beginning of the sentence into anywhere of the generated sentence. Though '<s>' seems helpless in unigram models, '</s>' is helpful. We include '</s>' in the pool thus give a chance for it to appear in the random sentence, and when it appears, we stop generating and end the sentence. Since each sentence contains an </s> as an end, suppose there are **n** sentences and **N** tokens in the corpus, there are **n** </s> tokens. Each time, the probability of choosing </s> is **n/N**. So if we denote the time we spent to get the first </s> as a random variable, it subjects to a Geometric Distribution, whose expectation is 1/probability of choosing '</s>' = 1/(**n/N**) = **N/n** = the average length of sentences in the training corpus. Therefore, end the random sentence with a '</s>' is reasonable.

In bigram and trigram models, the special tokens '<s>' and '</s>' perform better. When generating the first word of a random sentence in bigram and trigram models, we only consider the words that are actual beginning words of the sentences in the training corpus. The way to do this is to only pick up pairs(three in trigram models) whose first words are '<s>'. When ending random sentences, we also check to see if the second word(last word in trigram model) of the current pair is ‘</s>’. These two strategies simulated the actual beginning and ending of a sentence, thus making our random sentences more logical. In addition, we set a maximum length for a sentence in terms of the number of words in that sentence, to avoid random sentences that are too long.

PART 3 Smoothing & unknown words
In this section, we implemented a Good-Turing Smoothing for our language model for better estimation, as well as developed a methodology to handle the unknown words.

Panoramically, we introduce a special token <UNK> to represent unknown words. In order to better model the probability of <UNK>, we first retrain our model to assign proper probability mass to <UNK>. We can then treat unknown words (i.e. <UNK>s) as normal tokens and train our smoothed language model as usual.

We utilize the validation data set to spare an appropriate portion of probability mass for the newly introduced <UNK> tokens. Specifically, we treat the words occurred in the validation data but not the training data as unknown, and mark them as <UNK>. Subsequently, we combined the training data and the validation data as our new training data and train language models on it. Notice the new corpus contain <UNK> tokens, and by the assumption of independence of training data and validation data, it can approximate the real probability of unknown words. After this processing, <UNK> can be treated as an ordinary token, and further phases can proceed normally.

We then implement the Good-Turing Smoothing for our model. To compute Turing discount, we use another dictionary *SameFreqCount* to count the number of tokens of each frequency. For instance, *SameFreqCount[1]* stores the number of tokens that occur once in the training corpus. The following piece of codes constructs the *sameFreqCount* dictionary using the information stored in *count* dictionary (which we introduced in Part 1).

```
# turn count(key, freq) to
# sameFreqCount(freq, number of words with the same freq)
        sameFreqCount = Counter()
        for (key, freq) in count.items():
                sameFreqCount[freq] += 1
```

Then we derive the discounted frequency of each token following the standard simple Good-Turing estimation methodology. For simplicity, denote *SameFreqCount[i]* as n_i.
Then for each n_i, the revised count

$$n^*{}_i = (i+1)\frac{n_{i+1}}{n_i}$$

And for unseen tokens, i.e. $n^*{}_0$, we let

$$n^*{}_0 = \frac{n_1}{N}$$

where N again is the number of total tokens.

PART 4 Perplexity
In this part, we adhere to the formula given in the instruction and slides to calculate the perplexity of a test corpus.

$$PP = (\prod_{i=1}^{N} \frac{1}{P(w_i)})^{1/N}$$ for unigram models

$$PP = (\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})})^{1/N}$$ for bigram models

$$PP = (\prod_{i=2}^{N} \frac{1}{P(w_i|w_{i-1}w_{i-2})})^{1/N-1}$$ for trigram models

Note that the special tokens ‘<s>’ and ‘</s>’ are both counted as words.
Our result is as follows:

	bible_test on Bible corpus	hotel_test on Hotel Corpus
Unigram Model	280.896555128	348.412476117
Bigram Model	77.8783986212	47.9643416695
Trigram Model	24.2838511985	9.60870137394

As we can see from the table, as the ngram model we use gets more complex, the perplexity gets smaller, which means a better approximation of the actual corpus.

We choose to compute perplexity in a log-sum way. We turn the original formula into

$$logPP = -\frac{1}{N}\sum_{i=1}^{N} logP(w_i)$$ for unigram models

$$logPP = -\frac{1}{N}\sum_{i=1}^{N} logP(w_i|w_{i-1})$$ for bigram models

$$logPP = -\frac{1}{N-1}\sum_{i=2}^{N} logP(w_i|w_{i-1}, w_{i-2})$$ for trigram models

which prevents results from being overflown. (the base of all logarithms is 10)

When calculating the probability of the first word(words), we start with the token ‘<s>’ and end with ‘</s>’ instead of the actual beginning and ending words, e.g., in unigram models, w1 = ‘<s>’, wN = ‘</s>’, in bigram models, w0 = ‘<s>’, wN = ‘</s>’, and in trigram models, w0 = ‘<s>’, wN = ‘</s>’. We include these because we think sentence boundaries to be important in calculating perplexity.

Though the test corpus is read as a whole stream, we actually ‘separate’ it into individual sentences when calculating perplexity by ignoring the probability of the junction of two consecutive sentences where a token ‘</s>’ is preceded by a ‘<s>’. We think it meaningless to count this because we only care about the sequence of words within a sentence.

PART 5 Open-ended extension (Trigram Model)
We implement a trigram model as an extension. The main reason is that bigram model makes too strong an independence assumption that could undermine the expressiveness of the model. Trigram looks at longer history, thus is arguably more expressive.

Indeed, when examining our results on perplexity, one may notice that the perplexity of trigram model is significantly lower than that of bigram model on the same corpus, suggesting that the trigram model better captures the intrinsic characteristics of the corpus.

However, higher *n* is not preferable due to the data sparsity. Since our corpus are relatively small, any n-grams for n>3 suffer from severe sparsity problem and may result in poor performance.

Therefore, we choose trigram to implement, using the same mechanism of handling unknown words and smoothing as discussed in Part 3.

The extension can be done analogously to bigram. We collect additional counts for trigrams, and calculate probability as follows:

$$Prob(w|w_1, w_2) = \frac{count(w_1,w_2,w)}{count(w_1,w_2)}$$

Here the count refers to discounted count after Turing smoothing.

One thing to notice is that if both the numerator and the denominator are not seen before, we simply use $P = n^*_0$ to approximate this probability, as n^*_0 is the probability to estimate something unseen.

PART 6 Deception Detection
In this part, we use a simple way to judge whether a review is truthful or not. We do it in three steps:
(a) extract two sub-corpora from train and valid corpus, one only contains reviews that are truthful, the other one only contains reviews that are not truthful
(b) build good-turing language models on both sub-corpora
(c) compute the test reviews’ perplexities on both models and compare these two perplexities. The test review should belong to the model who gains a smaller perplexity on it.

We have done deception detection on both the hotel_test file and kaggle_test file. The results are shown below.

File	Model Type	Report Accuracy = Truthful reviews/All reviews
hotel_test	unigram	0.48275862069
kaggle_test	unigram	0.491666666667
hotel_test	bigram	0.528017241379
kaggle_test	bigram	0.425
hotel_test	trigram	0.85775862069
kaggle_test	trigram	0.466666666667

In step(a), we built a program called 'classifier.py' to deal with extraction. We read in all the lines of an original corpus(unpreprocessed), and then check every line to see if it begins with '1'. If it does, we write this line into a file which stores all truthful reviews. Otherwise, we write it to another file which stores all non-truthful reviews. Since Python reads lines by seeing '\n's, and every review in the original corpus is separated by a '\n' with no '\n's inside itself, every line that Python reads is an actual review in the corpus. We then repeat the same preprocessing procedures on the sub-corpuses as we did in Part 1, only except that we remove all the beginning '0's and '1's in every review. See clips of the program 'classifier.py' below.

```
    for line in trainfile.readlines():
            words = line.split()
            if len(words) == 0:
                    continue
            if words[0][0] == '0':
                    line = line[5:]
                    Nontruthful.write(line)
            if words[0][0] == '1':
                    line = line[5:]
                    Truthful.write(line)
```

In step(b), we just build the unigram, bigram and trigram models on both sub-corpora. Note that each corpus actually involves two files, one train file and another valid file. Our good-turing model uses both a train file and a valid file to build itself.

In step(c), we actually extract every single review in the test file to form a new individual file called 'test_clip'. In this way, our deception detector is taking these individual files as parameter one by one, with the availability to output judgement of every review while running.

EPILOGUE

The responsibilities are divided as follows.

Haotian Pan (hp343)	Random sentence generation; Perplexity; Deception Detection
Jian Jiang (jj544)	Preprocessing
Jue Zhang (jz578)	Unsmoothed ngrams; Smoothing & unknown words; Extension