N-Gram Language Models Documentation

Introduction

Throughout this documentation I will mainly be going over the points outlined in the project specifications. I will not go into many details regarding implementation as I feel I have mentioned these through the use of inline comments in the jupyter notebook which I provided wherever I saw fit.

Corpus Choices

I opted to use the Baby British Corpus and I decided to use an 80/20 split for my training and test corpus respectively.

Since the texts provided in the Baby British Corpus are organised into genres (fiction, newspapers, academic writing and spoken conversation) I decided to apply the 80/20 split to each genre rather than apply it to all the files together. Therefore, for example, from the 30 academic texts provided I extracted 7 of them (~20%) into the test corpus and the rest (~80%) into the training corpus. I decided to do this since I thought that there may be a difference in the vocabulary used in each genre, and hence I wanted to eliminate the possibility of extracting all or the vast majority of the test corpus from the same genre.

I implemented this into python by picking just over 20% of the files from each genre at random and inserting their names into a list. Then, when iterating through all the files in the Baby British Corpus, if the file name is present inside the list, the data is added to the test corpus and if not, it is added to the training corpus.

Below I have provided the size (word count) of the training and test corpus as well as the number of unique words found in both and the splits used.

```
calculate_corpus_stats()
```

```
Total number of words in the training corpus: 3920733
Unique number of words in the training corpus: 79046

Total number of words in the test corpus: 975307
Unique number of words in the test corpus: 37531

Test corpus %: 19.920323363371214
Training corpus %: 80.07967663662879
```

When loading the corpus, I am assuming that the “download” folder which includes all the texts is placed inside the same folder as the .ipynb file.

Language Models Build Time

The output of the codeblock below shows the amount of time it takes for all the three language models to build. This time also includes the time taken to extract the corpus and to build the frequency counts, which are both of course required to generate the language models.

```
start = time.time()

extract_corpus()
calculate_corpus_stats()
frequency_counter(training_corpus)
calculate_vanilla()
calculate_laplace()
calculate_unk()

end = time.time()
print(str(end-start) + " seconds")
```

```
56.39870071411133 seconds
```

Language Models Memory Allocation

Below I have provided the memory space required by each language model.

```
memory_vanilla = sys.getsizeof(P_vanilla_unigram) + sys.getsizeof(P_vanilla_bigram) + sys.getsizeof(P_vanilla_trigram)
memory_laplace = sys.getsizeof(P_laplace_unigram) + sys.getsizeof(P_laplace_bigram) + sys.getsizeof(P_laplace_trigram)
memory_unk = sys.getsizeof(P_unk_unigram) + sys.getsizeof(P_unk_bigram) + sys.getsizeof(P_unk_trigram)

print("Memory required for Vanilla Language Model Dictionary: " + str(memory_vanilla) + " bytes")
print("Memory required for Laplace Language Model Dictionary: " + str(memory_laplace) + " bytes")
print("Memory required for UNK Language Model Dictionary: " + str(memory_unk) + " bytes")
```

```
Memory required for Vanilla Language Model Dictionary: 128450840 bytes
Memory required for Laplace Language Model Dictionary: 128450840 bytes
Memory required for UNK Language Model Dictionary: 128450840 bytes
```

Sentence Probability Comparison

```
['<s>', 'it', 'was', 'a', 'nice', 'day', '</s>']
```

```
print("Vanilla: " + str(sen_probability(sentence, "vanilla", "bigram")))
print("Laplace: " + str(sen_probability(sentence, "laplace", "bigram")))
print("UNK: " + str(sen_probability(sentence, "unk", "bigram")))
```

```
Vanilla: 1.5871509696627568e-09
Laplace: 1.7437044228039855e-09
UNK: 1.5871509696627568e-09
```

Above can be seen the probabilities provided by each language model when the sentence “It was a good day” was provided to the sen_probability function. The probability provided by the vanilla and unk model is the same which I assume is because none of the words which I provided in the sentence were labelled with <unk>. Had I provided these kinds of words, I would have expected the unk model to produce a higher probability than that of the vanilla. The laplace model on the other hand provides a higher probability because of the smoothing operations performed.

Sentence Generation Comparison

```
In [28]: generate_sentence()

Select a Language Model:
Vanilla
Laplace
UNK
vanilla
Input a phrase
I think I should

I think I should think , how amazing , is that this is the news that paul will do now .
```

```
In [36]: generate_sentence()

Select a Language Model:
Vanilla
Laplace
UNK
laplace
Input a phrase
I think I should

I think I should think so .
```

```
In [40]: generate_sentence()

Select a Language Model:
Vanilla
Laplace
UNK
unk
Input a phrase
I think I should

I think I should be thought of her spare keys to addresses as the text of fonda 's book .
```

Overall, I do not think that there is much of a difference in the outputs provided by the different language models and I feel that all of them could have produced the output which they provided. In fact the first word generated by both the vanilla and the laplace is “think” which suggests that the probabilities of both are similar.

Testing

- One of the first tests I performed was to check if the frequency counts were correct. In order to test this, I initially populated the corpus with only one file from the english baby corpus. Since I did this, I could select a random unigram key from the frequency counts dictionary and check its corresponding frequency count. Then, I would open the file which was extracted into the corpus and use the find function (CTRL + F) to find the number of times the word appeared in the file. I am aware that this may be quite a naive approach, however it does the job and I managed to deduce that the frequency counts were correct by doing this.

- In order to ensure that my splits were as I wanted them, I used the code which can be seen below. Initially, I took exactly 20% of each of the files from each genre however I noticed that the splits were around 17% rather than the desired 20%, so I ended up adding more files to the test set and the splits are now as desired.

```
    print("Test corpus %: " + str(len(words_test) / (words_total) * 100))
    print("Training corpus %: " + str(len(words_training) / (words_total) * 100))
```

```
calculate_corpus_stats()
```

```
Test corpus %: 19.920323363371214
Training corpus %: 80.07967663662879
```

- In order to test if the corpus was extracted correctly, I simply printed the word count of the corpus after each iteration (after each file was extracted), from which I could notice that the word count was going up considerably after each iteration, and I therefore concluded that the corpus was being extracted correctly.

- In order to test the generate_sentences function, I printed out the variables which are in use, such as context (the final 2 words in the phrase inputted by the user) to ensure that the correct context was used to determine the next word in the sentence through the use of trigrams. I also chose to use trigrams as I found these to produce the best sentences, which makes sense considering that they would allow more context than the unigrams or bigrams. I also printed out the "probabilities" variable which acts as a weight to all the possible words that can be selected for the next word in order to determine if the weights were having an

effect, and I discovered that they were as most of the time the word with the highest or at least, a higher probability than most others was chosen.

- In order to determine if the <UNK> tokens replaced the required words in the training corpus, I repeatedly printed different sections of the training corpus and located every instance of the <UNK> token. I would then search the same section in the original training corpus and check what the actual word is. Then, I searched for the unigram counts of the word and found that each word which was replaced with a <UNK> token has a frequency of 2 or less, and I therefore deduced that the correct words were replaced. From there, I knew that my original algorithm for calculating the vanilla probabilities was correct, so I used the same code which would now iterate over the sentences in the updated corpus containing <UNK> tokens.