ASSIGNMENT1: LANGUAGE MODELING

s1825980

s1817788

TASK 1

For the `preprocess line` function, we originally define that the function of character **'?'** and **'!'** is the same as that of **'.'** in this situation, so we change them into character **'.'** at the beginning. And then remove all characters which are not English alphabet, space, digits, or the **'.'**. At the same time, we transform all digits to **'0'**, and lowercase all remaining characters. In view of some special cases, such as two spaces coming together which may be caused by the deletion before, we replace them with one space. Similarly, full-stop preceded by a space should also be replaced by an alone full-stop. In our view, this process will be closer to reality. Besides, in this function, we add a **STARTSYMBOL('##')** and a **STOPSYMBOL('#')** at the beginning and end of a sequence.

The python3 code of `preprocess line` is as follows.

```
import re
self.STARTSYMBOL = "##"
self.STOPSYMBOL = "#"
    def preprocess_line(self, line):

        line = re.sub('[?!]', '.', line)
        # remove all characters which are not English alphabet,
space, digits, or the ‘.’
        line = re.sub('[^A-Za-z0-9 .]', "", line)

        # transform all digits to '0'
        line = re.sub('[0-9]','0',line)

        # lowercase all remaining characters
        line = line.lower()

        # the process for some special cases
        line = re.sub('  ', ' ', line)
        line = re.sub(' \.', '.', line)
        # add a STARTSYMBOL('##') and a STOPSYMBOL('#')
        line = self.STARTSYMBOL+line+self.STOPSYMBOL
```

TASK 2

In terms of the pre-train language model provided `model-br.en` , even though we cannot be completely certain without seeing the corpus, we suppose that the model uses '**maximum likelihood estimation**' and '**add-one smoothing**' method. More specifically, for this method, the numerator is that the counts of trigrams plus one, and the denominator is the sum of the counts of the occurrences of previous two characters and the size of tokens' types(here is 30). The reason for this inference can be illustrated by evidence. For an example, for trigrams where both of previous two characters are space in this file, the probability of each trigram is 1/30, which is the same value with that calculated by the add-one estimation method. It is reasonable since it is nearly impossible under the English language rules for two space characters appearing at the same time. Thus, the probability is zero. But After add-one smoothing, it changes to the value 1/30.

TASK 3

We choose **Kneser-Ney smoothing method** to estimate the probabilities. Kneser-Ney smoothing method combines the ideas of several evaluation methods, including absolute discounting, interpolation and backoff. We suppose that it is more reasonable and can perform better here than these methods. Besides, Kneser-Ney method is more suitable to the small count. The equation of Kneser-Ney used to build the language model is

$$P_{KN}(w_i|w_{i-2},w_{i-1}) = \frac{max(0,C(w_{i-2},w_{i-1},w_i)-d)}{C(w_{i-2},w_{i-1})} + \lambda(w_{i-2},w_{i-1}) \cdot P_{KN}(w_i|w_{i-1})$$

Among them,

$$\lambda(w_{i-2},w_{i-1}) = \frac{d}{C(w_{i-2},w_{i-1})} \cdot |\{w : C(w_{i-2},w_{i-1},w) > 0\}|$$

And here we slightly changed the way to compute P_{KN} which can get a better result by using the training and testing set provided.

$$P_{KN}(w_i|w_{i-1}) = \frac{C(w_{i-1},w_i)}{C(w_{i-1})}$$

According to the Kneser-Ney smoothing, we have a basic assumption that words which have appeared in more contexts before are more likely to appear in a new context as well. To be concrete, In the training set, we subtract an absolute discount d from each count. Based on the result of a large number of experiments done by predecessors, we conventionally set the value of d to 0.75. In addition, λ is a regularization constant used to make the value of probability subtracted from the high-frequency words be assigned to the low-frequency words that may not appear before.

In this part, we choose dictionary to store the trigrams, bigrams, unigrams and other corresponding counts. And we can also use dictionary to store the estimation. The reason why we choose this data structure is that it is can be conveniently used to finding corresponding counts when computing.

Finally, an excerpt of the language model for English is as follows.

trigrams	**probability**
ng	8.205e-01
ng#	3.318e-04
ng.	2.530e-02
ng0	3.208e-06
nga	2.366e-03
ngb	3.208e-06
ngc	3.208e-06
ngd	3.010e-03
nge	8.848e-02
ngf	3.460e-04
ngg	9.053e-05
ngh	8.960e-04
ngi	2.428e-03
ngj	3.208e-06
ngk	3.208e-06
ngl	1.682e-03
ngm	3.208e-06
ngn	4.957e-04
ngo	6.366e-03
ngp	3.208e-06
ngq	3.208e-06

ngr	1.289e-02
ngs	2.005e-02
ngt	1.216e-02
ngu	2.444e-03
ngv	3.208e-06
ngw	3.208e-06
ngx	3.208e-06
ngy	1.190e-04
ngz	3.208e-06

From the excerpt above, we can find:

- The sum of all probability is 1, which is in line with the real situation.
- **'ng '** has the highest probability, which is reasonable. Because it is actually a common trigram. For example, the last two characters of morpheme **'-ing'** is **'ng'**
- Those who have the lowest probability(*3.208e-06*) does not appeared in training file, but are also assigned probability by smoothing.

In conclusion, these trigrams and corresponding probabilities are what we expect for.

TASK 4

In order to use a language model to generate random output sequences, we decide to start from "**##**" which means the beginning of paragraph and generate the third character after "**##**". Then we use the middle and last character in this generated trigram to generate the next character. Therefore, we can generate *N* characters by loop N-2 times(the first two characters are "**##**").

We use `cumsum` and `digitize` functions provided by `numpy` to randomly generate a character based on a certain bigram. `cumsum` function is used to get the cumulative sum of probabilities at each index. Then use `digitize` function and a random float number between 0 and 1 to get an index of the key of the generated trigram.

The pseudocode is as follows:

```
/* make sure that keys and values will use the same ordering */
outcomes <- np.array(keys)
probs <- np.array(values)
/* make an array with the cumulative sum of probabilities at each
index */
bins <- np.cumsum(probs)
/*
create a random float number from 0-1, then use digitize to get
index.
the digitize fuction tells us which bin the random number fall into.
use the index to find the corresponding trigram, and the last
character is what we */
want to generate
generate <- outcomes[np.digitize(np.random.rand(), bins)][2]
```

Here is the 300 characters of random output generated by:

- the model we estimated from the English training data

> ##shaver eur precture se not or shorkin of wout is i witency totinal agared forions mr ame of port i migh muct is all antedeves.###i hato thateursh cust as ted an asues whousequal europed drandeball thanint govate clerm the stradany ime a mr i wor ounspore mosat of of anal agral prin to han thell mart

- and the model in `model-br.en`

> ##that thats pre put are juillock.###her boodys uke fl.###what yourty gon ing momes onq0lt a.###ank is the he back of th.###thats gooks.###loo my.###backs.###te.###whair.###all re trplay.###put.###calk.###what drand.###a sit.###do you wase bight thats that you da box.###bye.###nothis na lo.###call go pare do hat.###okay.###be sits am.###y

We suppose that there are '**##**' at the beginning of paragraph and a '**#**' at the end of paragraph. So '**###**' represents the start of a new paragraph. By observing these two sequences, we can find that the sequence 2 has much more '**###**' and each sentence between the two '**###**' is much shorter than the sequence 1. The reason for such a situation is the probability of trigram keys which contain '**#**' in `model-br.en` is higher than that in the model we generated. It is obvious that '**#**' is always followed by '**.**' since it represents the end of paragraph. Thus we can deduce that `model-br.en` is generated by the kind of train file whose paragraphs are very short or even just contain one short sentence, like poems or lyrics, which is different from the model we generated.

TASK 5

The values of the perplexity which is computed by the test document provided under English, German and Spanish language models are approximately ***8.95***, ***31.65*** and ***30.54*** respectively. Compared with the perplexity of these three models, we can find that the perplexity under English model is much less than other two models. Therefore, from the figure, it can be inferred that the test document is more likely to be in English rather than German or Spanish.

In order to determine whether a test document is written by English, we come up with a method that setting a perplexity threshold by testing file and we assume that the file with the perplexity below the threshold is an English file. Based on the three values of perplexity on different language models(English, German and Spanish) we got above, we believe ***10*** is a reasonable threshold.

In order to verify our assumption, we tested two different files under the English language model. One document file is an English text from the Internet (https://www.gutenberg.org/files/58102/58102-0.txt), and the other one is generated by function `generate_from_lm` based on the English model. Both files have approximately 480,000 characters.

As the result, these two files have similar values of perplexity, which are ***9.22*** and ***5.71*** respectively, and both of them are lower than ***10***. So it is the same with our expectations.

However, from the **task 4** we can know that the second file which generated by English LM does not belong to any natural language but has a smaller value of perplexity. Therefore, even though we can estimate wether a test document is written by English by this method, we cannot assert it. In our view, there are three main reasons for this result:

1. We use very few samples to generate the English model and estimate the perplexity under this model(only one training file and one testing file). Besides, many trigrams do not appeared actually according to the English grammar rules but get the probability by smoothing, so we cannot make sure that this model can be used on every kind of English documents.
2. To set this threshold, we only compared the performance of test file on three language models(English, German and Spanish). However, there may be other languages which are different from these three languages. If the values of the perplexity of those languages' documents were low and similar to that of English, thus it would be hard to distinguish those languages.
3. For those who are not written by natural language but generated buy specific rules, they may also meet the demand and we cannot distinguished them only by perplexity.

TASK 6

From the tasks above, we found that the trigram language model over characters is not good enough to generate natural English text. Therefore, we put forward a question that **What is the best N in N-gram language model over characters in order to generate correct and abundant English text**.

Obviously, when *N* is small, the generated characters cannot consider the part of the same word that is beyond *N*, so it is difficult to generate accurate words, and it is impossible to consider the relationship between words and words. On the contrary, the condition(w_1, w_2 ..... w_n) will be quite long when *N* is a large number, which increase the difficult on calculations and statistics. Moreover, the probability for each N-gram is too small to make the model deal with various text.

In our opinion, one way to solve this problem is to generate different N-gram language models using the same estimation method and sufficient training sets, and then compare the values of perplexity computed by these different models under a large number of test sets. In order to reduce the complexity, we can first let the number of *N* increases by 5 from 5 to 50, and then observe the changing trend of perplexity and find the interval with the least perplexity. In this interval, we test each different values of *N* in turn to find the best value.

However, we cannot get the best result. Either value of perplexity we got from the test set or the relevant generated sequences are cannot meet the demand. We suspect that the main reason for this result is that the sample in the test set is insufficient. As *N* increases, the demand for the number of samples increases exponentially. Insufficient sample size may cause most of n-grams to appear only a few times or even not at all in the test set, and thus had a significant impact on the results.

In addition, we also come up with another question: "**Can we combine trigrams with grammar rules?**". It means that we count each trigram based on the part of speech. When calculating the perplexity, we first determine the part of speech of the word, and then find the trigram in the statistics of this part of speech. At the same time, we use the same way to generate the sequence, and we shall determine the part of speech of the next word before generating the next word. We believe that there are some of benefits as follows:

- The frequency of trigrams in the different part of speech is different. So the factor of part of speech should be taken into account to get a more accurate result.
- The generated words will be more proper for the rules of grammar.
- Combined with grammar, the text which does not belong to the language we used to train our model can be easier to identify.

We will deal with this problem in further studying and experiments.