IFT6285 Traitement Automatique des Langues Naturelles
Professor: Philippe Langlais
Student: Gustavo Alonso Patino Ramirez

Homework 1

1. Owing to the training corpus is very large for my computational capabilities, it was decided to read only a part of the training corpus and create a subset of the corpus consisting of $1e7$ words. The training corpus was tokenized using the *word-tokenize* in *nltk* method of python. It was implemented the function **build-unigrams** (see appendix) to map each word in the corpus and its frequency in the document. Using this function it was determined that the training corpus has 63011 different words. Through the map obtained from **build-unigrams**, it can also be inferred that the training corpus has 7359 words seen at least 100 times. The test corpus was also tokenized concluding that the test corpus has 1797 words that do not exist in the training corpus. Words with a frequency greater than 100 and the test words not seen in the training were implemented in the function **count-words**.

2. For each word in the training corpus ($prefix_i$), it was created a map of bigramas with the function **build-bigram** . The map has the following format:

$$\begin{aligned}&\{prefix_1 : \{suffix_{11} : frequency,\ suffix_{12} : frequency, ...suffix_{1m} : frequency\},\\ &\ prefix_2 : \{suffix_{21} : frequency,\ suffix_{22} : frequency, ...suffix_{2m} : frequency\},\\ &\ \vdots\\ &\ prefix_n : \{suffix_{n1} : frequency,\ suffix_{n2} : frequency, ...suffix_{nm} : frequency\}\}\end{aligned} \tag{1}$$

   while the unigrams (according to the function **build-unigrams**) of

the corpus have the format:

$$\begin{aligned} &\{word_1 : frequency, \\ &\ word_2 : frequency, \\ &\ \vdots \\ &\ word_n : frequency\} \end{aligned} \tag{2}$$

In this way the training of our model consists in the construction of the unigrams and bigrams of the corpus. The computational time spent in training was 54.1

3. To make certain comparisons between the training corpus and the corpus test it is necessary to do a pretreatment of the training body and define unknown words. In order to do this task, we slice along the tokens of the training corpus and use each token as a key to our map of unigrams, if the word appears with a low frequency (arbitrarily defined as 10) we consider that token as unknown "UNK", with the new set of tokens the model is trained again, it means, a new map of unigrams and bigrams of the training body is created, but this time with unknown words.

4. The perplexity of our bigram system is defined as

$$PP(W) = \log\left(\prod_{i=1}^{n} \frac{1}{P(w_i \mid w_{i-1})}\right)^{1/n} \tag{3}$$

and implemented in the function **perplexity**. To get the probability we proceed at follows: we take the word w_{i-1} (prefix) and the word w_i (suffix) and use them as keys in our map of bigrams of the training corpus, using the 2 keys we obtain the value of how many times the suffix follows the prefix in our training, and we divide this value by the number of times the prefix appears in the training body (using the map of training unigrams and the prefix as the key). It should be noted that both the prefix and suffix in this case corresponds to words in our test corpus and the unigrams and bigrams of our training. This probability is calculated in **get-probability**, and a value of 133.18 was obtained for the perplexity.
Thanks to the function **random-limits-generator** we can generate random pieces of our test and evaluate their perplexity using the function **perplexity-random-test**. For 10 different and random pieces

of our test we obtained an average value for the perplexity of 130.167 with a standard variation of 0.322, these calculations were done inside the same function.

5. The model is able to make word predictions in the test using the maps of unigrams and bigrams in the training corpus. A random word was taken from the test ("need" in this case) and the most probable suffixes of this word were evaluated using the function **prediction-words**, obtaining this set of probabilities:
need –> [('to', 0.530083), ('for', 0.154527), ('a', 0.051397), (',', 0.017706), ('the', 0.014139), ('of', 0.013888), ('more', 0.013342), ('is', 0.011496), ('.', 0.010825)]

   In order to evaluate the performance of the model, the unigrams and bigrams of the test corpus were created. From the bigrams of the test we see that the words that really should follow a "need "with probabilities in the test are:

   need –> [('to', 0.483871), ('for', 0.193548), ('.', 0.032258), ('locally', 0.032258), ('the', 0.032258), ('reassurance', 0.032258), ('your', 0.032258), ('stability', 0.032258), ('a', 0.032258)]
   Although the model did not predict all the words of the test (hit 5 of 9) it is notable that it was able to guess the 2 most likely words, despite the training corpus being relatively small due to computational limitations.

6. Using the unigrams of the training corpus we can calculate the top of the most probable words in the test. For this we slice through each of the words of the test and evaluate its frequency in the training corpus and thus obtain the probability. According to the function **unigram-test-prob** the most probable words in the test are:

   [('of', 0.0312), ('and', 0.0245), ('in', 0.01931), ('that', 0.0154), ('is', 0.0147), ('a', 0.0145), ('UNK', 0.0131), ('for', 0.0096), ('I', 0.0096)]
   According to the test unigrams the words that should be more likely are: [& , of, @,'a', 'and', 'quot', 'in', 'apos', 'is']

   Again, the model was able to match 5 of the 9 most probable words in tests, being remarkable that one of the most probable words for the model (in the second place) is just the most probable word of the test.

At follows you can see some pieces of the code, a complete version of the code was not provided because the document exceeds the size allowed. A complete version can be sent if required.

APPENDIX

```
import numpy as np
import matplotlib.pyplot as plt
import nltk
import time

#build an array of tokens from the text
def build_tokens(f):
    tokens = nltk.word_tokenize(f.read())
    f.close()
    return tokens


# given a list of tokens, return a dictionary (the unigram) where key
# are the existing unique tokens and values are their counts.
def build_unigrams(tokens):
      unigram_table = {}

      for token in tokens:
          if token in unigram_table:
                unigram_table[token] += 1
          else:
                unigram_table[token] = 1

      return unigram_table


def build_bigram(tokens):
    bigram_table={}
    num_bigrams=0

    for i in range(len(tokens)-1):
        if tokens[i] in bigram_table:
            if(tokens[i+1] in bigram_table[tokens[i]]):
               bigram_table[tokens[i]][tokens[i+1]] += 1
            else:
               bigram_table[tokens[i]][tokens[i+1]] = 1
               num_bigrams+=1

        else:
            bigram_table[tokens[i]]={}
            bigram_table[tokens[i]][tokens[i+1]] = 1
            num_bigrams+=1

    return (bigram_table, num_bigrams)

def get_probability(prefix, suffix, unigram, bigram):
    count = 0

    if suffix in bigram[prefix]:
        count = bigram[prefix][suffix]
    if count < 5:
        count = 1

    return (round(count / (unigram[prefix]),6))

def perplexity(test_token,unigram_unk, bigram_unk):
    length = len(test_token)
    for i in range(length):
        if test_token[i] not in unigram_unk:
            test_token[i] = 'UNK'

    sum = 0.0
    for i in range(length - 1):
        prob = get_probability(test_token[i], test_token[i+1], unigram_unk, bigram_unk)
        sum+=((np.log(prob))*(-1))

    return np.exp((sum / length))
```

```
def random_limits_generator(tokens_test):

    while(True):
      limits=np.random.randint(0, len(tokens_test),2)
      if(np.abs(limits[1]-limits[0])>=7000): break
    return(np.sort(limits))


def perplexity_random_test(tokens_test,unigram_unk, bigram_unk):
    perp=[]
    for i in range(2):
       print("step: ",i)
       limits=random_limits_generator(tokens_test)
       tokens_random=tokens_test[limits[0]:limits[1]]
       p=perplexity(tokens_random,unigram_unk, bigram_unk)
       perp.append(p)
    return (round(np.mean(perp),3), round(np.std(perp),3))

#random word in test and predict the most probable suffixes of the word
def prediction_words(tokens_test, unigram_unk, bigram_unk):
    symbols=[".",",","?","the","to","-",";"]
    L=[]
    while(True):
      rn=np.random.randint(0, len(tokens_test))
      random_word_test=tokens_test[rn]
      if(random_word_test not in symbols):
          break

    neighbors=bigram_unk[random_word_test]
    table_prob={}

    for neigh in neighbors:
         prob=get_probability(random_word_test, neigh, unigram_unk, bigram_unk)
         if(prob>=1e-5):
           table_prob[neigh]=prob

    for key, value in sorted(table_prob.items(), key=lambda item: item[1]):
         t=(key,value)
         L.append(t)

    return random_word_test, L[len(L):len(L)-10:-1]


 def unigram_test_prob(test_token, unigram_train):
    test_token_local=test_token
    symbols=[".",",","?","the","to","-",";"]
    length=len(test_token_local)
    for i in range(length):
        if test_token_local[i] not in unigram_train:
            test_token_local[i] = 'UNK'

    unigram_test=build_unigrams(test_token_local)
    max_prob={}
    for word in unigram_test:
         if(word not in symbols):
           prob=(unigram_train[word])/len(tokens_train)
           max_prob[word]=prob

    L=[]
    for key, value in sorted(max_prob.items(), key=lambda item: item[1]):
         t=(key,value)
         L.append(t)

    return L[len(L):len(L)-10:-1]

def count_words(unigrams_train, unigram_test):
    words_100=0
    for key in unigrams_train:
       if(unigrams_train[key]>=100):
           words_100+=1

    words_non_vu=0
    for key in unigram_test:
        if(key not in unigrams_train):
           words_non_vu+=1
    return (words_100, words_non_vu)
```