Spelling Correction

Chen, Aoxue

February 23, 2019

1 Preparation

As the text is in category of news, we can use the Reuters corpus in **ntlk**.

First, we extract the word frequency, bigram and other informations from the corpus. To save time, we can use **pickle** library to save the data into an external file.

```
In [1]: import numpy as np
        import pandas as pd
        from pandas import Series, DataFrame
        import math
        import nltk
        from nltk import word_tokenize
        from nltk.corpus import reuters
        import pickle
```

```
In [2]: reuters_list = reuters.words()
        un_freq = nltk.FreqDist(reuters_list) # word frequency
        bigrams = nltk.bigrams(reuters_list)
        bi_freq = nltk.FreqDist(bigrams) # Bigram frequency
        V = len(un_freq) # vocabulary size
        N = len(reuters_list) # tokens
```

Save the data into dataFile:

```
In [3]: with open('dataFile','wb') as fw:
            pickle.dump(un_freq,fw)
            pickle.dump(bi_freq,fw)
            pickle.dump(V,fw)
            pickle.dump(N,fw)
```

Open a new python file for main program. Firstly open the vocabulary file:

```
In [4]: with open('vocab.txt') as vocab_file:
            lines = vocab_file.readlines()
        vocab = [line.strip() for line in lines]
```

Load the saved data from dataFile:

```
In [5]: with open('dataFile','rb') as fr:
            un_freq = pickle.load(fr)
            bi_freq = pickle.load(fr)
            V = pickle.load(fr)
            N = pickle.load(fr)
```

Load the test data:

```
In [6]: testdata = pd.read_table('testdata.txt', header=None)
        n = testdata.shape[0]
```

2 Generate the candidate words

If we calculate each word's edit distance to every word in the vocaburary, it will take a huge amount of time. Thus we can first generate a set of possible words with edit distance of 1 to each misspelled word, then check if they are legal.

```
In [8]: letters = "abcdefghijklmnopqrstuvwxyz"
        letters_upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        other = " -'"

        def distance_1(word):
            total = {word} #Including itself
            for i in range(0,len(word)):
                l = word[:i]
                r = word[i:]
                alphabet = letters
                if len(word) > 1 and i > 0:
                    total.update(set([l[:i-1]+r[0]+l[i-1]+r[1:]])) #Transposition
                    if i < len(word):
                        alphabet = letters+other # Punctuation cases
                total.update(set([l+c+r for c in alphabet]+[l+r+c for c in alphabet]))
                #Insertion
                total.update(set([l+r[1:]])) #Deletion
                total.update(set([l+c+r[1:] for c in alphabet])) #Substitution
            total = total - {""} #Remove empty string
            return total
```

For words with uppercase letter, we can take the short cuts: * If the word contains one uppercase letter that is not in the initial position, we can try to swap the letter with the current initial letter; * If the word only have one lowercase letter, we can consider deleting the letter or substitude the letter with an uppercase one; * In other cases, if the word have more than one uppercase letter, it's reasonable to insert or substitute an uppercase letter into the misspelled word; * If none of the mentioned operation can generate a legal word, then take the word as in the normal situation.

```
In [9]: def find_upper(word): #Output the number and position of uppercase letters
            count = 0
            pos = []
```

```
    for i in range(0,len(word)):
        if word[i].isupper():
            count += 1
            pos.append(i)
    return [count,pos]

def uppercase_corr(word):
    total = {word}
     #UPPERCASES
    find = find_upper(word)
    num = find[0]
    ind = find[1]
    if num == 1 and ind[0] == 0:
        return {word}
    elif num == 1 and ind[0] > 0: #One uppercase
        new_word = word[ind[0]]+word[1:ind[0]]+word[0]+word[ind[0]+1:]
        return {new_word,word}
    elif num == len(word)-1: #e.g. INTERNAsIONAL
        for i in range(0,len(word)):
            if i not in ind:
                l = word[:i]
                r = word[i:]
                alphabet = letters_upper
                if len(word) > 1 and i > 0:
                    if i < len(word):
                        alphabet = letters_upper+other # Punctuation cases
                total.update(set([l+r[1:]])) #Deletion
                total.update(set([l+c+r[1:] for c in alphabet])) #Substitution
                break
        return total
    else: #e.g. INTERVNTION ltGR
        for i in range(0,len(word)):
            l = word[:i]
            r = word[i:]
            alphabet = letters_upper
            if len(word) > 1 and i > 0:
                #total.update(set([l[:i-1]+r[0]+l[i-1]+r[1:]])) #Transposition
                if i < len(word):
                    alphabet = letters_upper+other # Punctuation cases
            total.update(set([l+c+r for c in alphabet]+[l+r+c for c in alphabet]))
            #Insertion
            #total.update(set([l+r[1:]])) #Deletion
            total.update(set([l+c+r[1:] for c in alphabet])) #Substitution
        return total
    return total
```

If still no legal word is generated, we must consider the situation that the edit distance is more than 1.

When the edit distance is 2, we have:

```
In [10]: def distance_2(word):
             return set(d2 for d1 in distance_1(word) for d2 in distance_1(d1))
```

However, it's easy to find the lack of efficiency in this method. Observing the testing text, we find that the situation where edit distance is more than 1 are mostly just swapped two of the letters in the word. So we can use the function below instead:

```
In [11]: def non_distance_1(word): #Swap
             total = set()
             for i in range(0,len(word)-1):
                 for j in range(i+1,len(word)):
                     if word[i] != word[j]:
                         new_word = word[:i]+word[j]+word[i+1:j]+word[i]+word[j+1:]
                         total.add(new_word)
             return total
```

3 Calculate the probability

We use the frequency of unigrams and bigrams to approximately calculate the emerging probability of the next word, to choose a word most likely to appear.

```
In [12]: def log_smoothed_prob(pre,next):
             prob = math.log10((bi_freq[(pre,next)]+1))-math.log10((un_freq[pre]+V))
             return prob
```

To prevent underflow, we take the logarithm of the probabilities. In addition, use Laplace Smothing to roughly handle the zeros.

4 Correction: non-word

```
In [13]: def correction(pre,word,vocab): #take the previous word as parameter as well
             prob_dict = dict()
             find = find_upper(word) # find uppercase
             num = find[0]
             if num > 0: #with uppercase
                 for y in uppercase_corr(word):
                     if y in vocab:
                         prob_dict[y]=log_smoothed_prob(pre,y)
             for y in distance_1(word): #normal situation
                 if y in vocab:
                     prob_dict[y]=log_smoothed_prob(pre,y)
             for y in non_distance_1(word): #swapping situation
                     if y in vocab:
                         prob_dict[y]=log_smoothed_prob(pre,y)
             if len(prob_dict) == 0:
                 return word #if still no candidates generated, give up
```

```
            else:
                return max(prob_dict,key=prob_dict.get)
                #output the one with greatest probability
```

Now we can use the functions to correct the words!
Meanwhile, record the wrong words we can't detect at the moment for further investigation.

```
In [17]: exist_real_word_errors = list()
         result = testdata.drop(columns=1)
         for i in range(0,n):
             non_word_count = 0
             sentence = word_tokenize(testdata[2][i])
             for p, word in enumerate(sentence):
                 if non_word_count == testdata[1][i]:
                     #no need to loop when the number is enough
                    break
                 if word not in vocab:
                     non_word_count += 1
                     correct_word = correction(sentence[p-1],word,vocab)
                     if i<20: #only print the head
                         print(str(i+1)+" "+word+" "+correct_word)
                     result.iat[i,1] = result.iat[i,1].replace(word,correct_word)
             if non_word_count != testdata[1][i]:
                 exist_real_word_errors.append(i) #real word error positions
```

```
1 protectionst protectionist
2 Tkyo Tokyo
3 retaiation retaliation
4 tases taxes
5 busines business
7 Taawin Taiwan
8 seriousnyss seriousness
9 aganst against
10 bililon billion
11 sewll swell
12 importsi imports
13 Sheem Sheen
14 wsohe whose
15 Koreva Korea
16 Japn Japan
17 semicondctors semiconductors
18 advantagne advantage
19 Lawrenc Lawrence
```

We get the position with read word errors:

```
In [19]: print(exist_real_word_errors)
```

```
[5, 19, 47, 54, 64, 118, 123, 138, 153, 157, 170, 177, 265, 267, 294, 356, 391, 410, 435, 471, 4
```

5 Correction: real-word

We first instruct a function to generate candidate sentences:

```
In [18]: def generate_candidates(sentence):
             candidates = list([sentence]) #contains itself
             for p,word in enumerate(sentence):
                 if word[0] not in letters+letters_upper:
                     #let go of the one with punctuations at initial
                     continue
                 elif word[0] in letters_upper: # uppercase initial
                     for replace in distance_1(word.lower()):
                         if len(replace) == 1: # one letter word
                             replace = replace[0].upper()
                         else:
                             replace = replace[0].upper()+replace[1:]
                         if replace in vocab:
                             candidate = sentence[:p]+[replace]+sentence[p+1:]
                             if candidate not in candidates: # avoid repitition
                                 candidates.append(candidate)
                 else: #normal
                     for replace in distance_1(word):
                         if replace in vocab:
                             candidate = sentence[:p]+[replace]+sentence[p+1:]
                             if candidate not in candidates:
                                 candidates.append(candidate)
             return candidates
```

Calculate the approximate probability of each possible sentences:

```
In [20]: def sentence_prob(sentence):
             log_prob = 0
             for p,word in enumerate(sentence):
                 if p == 0:
                     log_prob = math.log10(un_freq[word]+1) - math.log10(N+V)
                 else:
                     pre = sentence[p-1]
                     new_prob = log_smoothed_prob(pre,word)
                     log_prob = log_prob + new_prob
             return log_prob
```

Selecting the most proper sentence in the candidates:

```
In [21]: def real_word_correction(sentence):
             max_prob = -10000
             best_candidate = sentence
             for candidate in generate_candidates(sentence):
                 prob = sentence_prob(candidate)
                 if prob > max_prob:
```

```
                    max_prob = prob
                    best_candidate = candidate
            return best_candidate
```

Main program:

```
In [22]: for i in exist_real_word_errors:
            sentence = word_tokenize(result.iat[i,1])
            correct_sentence = real_word_correction(sentence)
            for j in range(0,len(sentence)):
                if sentence[j] != correct_sentence[j]:
                    word = sentence[j]
                    correct_word = correct_sentence[j]
                    result.iat[i,1] = result.iat[i,1].replace(word,correct_word)
                    break
            if i<300:
                print(str(i+1)+" "+word+" "+correct_word)
```

```
6 pace place
20 whoe whole
48 alter after
55 taking making
65 trade traded
119 so to
124 mouth month
139 latter later
154 boots boost
158 rule ruled
171 trades trade
178 sill still
266 consume consumer
268 markets market
295 stacks stocks
```

Write the corrected sentences into result.txt:

```
In [23]: np.savetxt('result.txt',result.values,fmt='%s',delimiter='\t',)
```

6 Evaluation

Use **eval.py** to calculate the accuracy:

```
In [24]: import nltk
         anspath='./ans.txt'
         resultpath='./result.txt'
         ansfile=open(anspath,'r')
         resultfile=open(resultpath,'r')
```

```
        count=0
        for i in range(1000):
            ansline=ansfile.readline().split('\t')[1]
            ansset=set(nltk.word_tokenize(ansline))
            resultline=resultfile.readline().split('\t')[1]
            resultset=set(nltk.word_tokenize(resultline))
            if ansset==resultset:
                count+=1
        print("Accuracy is : %.2f%%" % (count*1.00/10))
```

```
Accuracy is : 96.80%
```