1. Have Fun With Regular Expression

1.1 On exercises from SLP

2.1.2 The set of all lower case alphabetic strings ending in a b;

In [62]:

```
import re
```

In [17]:

```
pattern1 = r'^[a-z]*b$'
```

- ^: starting point
- [a-z]* : lower case alphabets that can have a length of 0 or more
- b$: string ends with a "b"

In [18]:

```
#test cases
print(re.search(pattern1, '2ab')) #shouldn't work because it contains int
print(re.search(pattern1, 'Lab')) #shouldn't work because of the upper case 'L'
print(re.search(pattern1, 'lab')) #should work
print(re.search(pattern1, 'b')) #should work
```

```
None
None
<re.Match object; span=(0, 3), match='lab'>
<re.Match object; span=(0, 1), match='b'>
```

2.1.3 The set of all strings from the alphabet a, b such that each a is immediately preceded by and immediately followed by a b;

In [19]:

```
pattern2 = '^(b*ba)*bb*'
```

- ^: starting point
- ba: "a" is immediately preceded by a "b"
- the first * : it doesn't matter how many "b"s are before a
- the second * : it doesn't matter if a exists
- the "b" after the second * : "a" is immediately followed by a "b"
- the b* at the end: it doesn't matter how many "b"s are after a

In [20]:

```
#test cases
print(re.search(pattern2, 'aba')) #shouldn't work because a is not preceded by b
print(re.search(pattern2, 'b')) #should work
print(re.search(pattern2, 'bab')) #should work
print(re.search(pattern2, 'bbbabbbbabbabbbbbbbbbabb'))#should work
```

```
None
<re.Match object; span=(0, 1), match='b'>
<re.Match object; span=(0, 3), match='bab'>
<re.Match object; span=(0, 23), match='bbbabbbbabbabbbbbbbbbabb'>
```

2.2.1 The set of all strings with two consecutive repeated words (e.g., “Humbert Humbert” and “the the” but not “the bug” or “the big bug”);

In [21]:

```
pattern3 = r'([a-zA-Z]+)\s+\1'
```

- ([a-zA-Z]+) : a stored word
- \s+: spaces
- \1: repetition of the stored word

In [23]:

```
#test cases
print(re.search(pattern3, 'the bug')) #shouldn't work; no repetition
print(re.search(pattern3, 'the big big bug')) #should work
```

```
None
<re.Match object; span=(4, 11), match='big big'>
```

2.2.2 All strings that start at the beginning of the line with an integer and that end at the end of the line with a word;

In [25]:

```
pattern4 = r'^\d+[^\n]*[a-zA-Z]+$'
```

- ^\d+: begining of the line with an integer
- [^\n]*: anything except a new line
- [a-zA-Z]+$: end the line with a word

In [26]:

```
#test cases
print(re.search(pattern4, ' 4559 STAT ')) #shouldn't work; it starts and ends wi
th space
print(re.search(pattern4, '4559@#%$$%\n^STAT')) #shouldn't work ; contains a new
line
print(re.search(pattern4, '4559@#%$$%^STAT')) #should work
```

```
None
None
<re.Match object; span=(0, 15), match='4559@#%$$%^STAT'>
```

1.2 On a regex for phone numbers

In [30]:

```
phone_number = r'^((\d{3}-)|(\(\d{3}\)\s)?(\d{3})-(\d{4})$'
```

- ((\d{3}-)|((\d{3}))\s) : either (xxx) or xxx-
- ?: The area code can be ignored
- (\d{3})-(\d{4}): the seven number digit
- ^ and $: match the pattern strictly

In [31]:

```
#test cases

print(re.search(phone_number, '-456-7890')) #shouldn't work; not a phone number

#all should work below
print(re.search(phone_number, '(123) 456-7890'))
print(re.search(phone_number, '123-456-7890'))
print(re.search(phone_number, '456-7890'))
```

```
None
<re.Match object; span=(0, 14), match='(123) 456-7890'>
<re.Match object; span=(0, 12), match='123-456-7890'>
<re.Match object; span=(0, 8), match='456-7890'>
```

1.3 On a regex for reading in files

In [15]:

```
file_pattern = r'UVA_(5[0-9]|60)_(\d{3})_F_90_(uninj|inj)_(y|n)\.csv'
```

The file pattern is based on the format of the file names. Specifically,

- UVA: school name
- (5[0-9]|60): age range [50,60]
- (\d{3}): the id consists of 3 digits according to the files.
- F: female only as required
- 90: survey 90 as required
- (uninj|inj): it doesn't matter whether the person is injured or not.
- (y|n): it doesn't matter yes or no to the question.

Now let's test it out.

In [64]:

```
import os
files = os.listdir('./regex_files')
ct = 0
for filename in files:
    if re.search(file_pattern, filename) != None:
        ct+=1
        print(filename)
print(f'Total number of files found: {ct}')
```

```
UVA_50_850_F_90_uninj_n.csv
UVA_50_476_F_90_uninj_n.csv
UVA_50_436_F_90_inj_n.csv
UVA_59_351_F_90_inj_y.csv
UVA_56_975_F_90_inj_y.csv
UVA_52_678_F_90_uninj_y.csv
UVA_59_167_F_90_inj_n.csv
UVA_58_684_F_90_inj_y.csv
UVA_57_462_F_90_inj_n.csv
UVA_53_542_F_90_uninj_y.csv
UVA_58_760_F_90_inj_n.csv
UVA_53_464_F_90_uninj_n.csv
UVA_60_352_F_90_uninj_y.csv
UVA_54_149_F_90_uninj_y.csv
UVA_60_297_F_90_inj_y.csv
UVA_50_990_F_90_inj_y.csv
UVA_60_285_F_90_uninj_n.csv
UVA_58_635_F_90_inj_y.csv
UVA_59_423_F_90_inj_y.csv
UVA_50_681_F_90_uninj_n.csv
UVA_56_498_F_90_inj_n.csv
UVA_55_741_F_90_inj_y.csv
UVA_59_479_F_90_uninj_y.csv
UVA_58_834_F_90_inj_n.csv
UVA_59_795_F_90_inj_n.csv
UVA_56_759_F_90_inj_n.csv
UVA_52_107_F_90_uninj_y.csv
UVA_58_198_F_90_inj_y.csv
UVA_57_617_F_90_inj_n.csv
Total number of files found: 29
```

2. Our First Language Model

In [2]:

```
import pandas as pd
bigrams =  pd.read_csv('bigrams.csv')
unigram = pd.read_csv('unigrams.csv')
```

2.1 On the dataset side: discuss the potential pitfalls with using this specific data set to train the bigram model

In our case, by using only the first 80 sentences of the book as corpus, we are more than likely to get a biased language model because the corpus is very likely not to contain all possible words, especially words appeared not in literature but in sports, medication, engineering, law, etc.

One of the consequences is that the model will be subject to **the problem of sparsity**. Because the corpus is limited, words that do occur in the test set do not occur in the training set, and when we apply the model based on such corpus to test data, we will have many cases of putative zero probability bigrams that should really have non-zero probability.

For example, we have the following bigrams in the corpus:

- I said
- said what
- what are
- are you
- you doing

And suppose our test set has a phrase:

- You said

Our model will incorrectly estimate the probability P(You said) = P(said|you) *P(you) = 0* 2/5 = 0.

What's more, we will get wrong estimation for sentences if the probability of any bigrams in the test set is 0 because the probability of a sentence can be factorized into products of probabilities of bigrams.

While the sparisity is about the problem of words whose bigram probability is zero, The **out of vocabulary(OOV)** problem is that we may have to deal with words we haven’t seen before because of the limited corpus. There are several solutions to solve this problem, such as inserting them OOV as pseudo-words "UNK" into training data and replacing words in the training set "UNK" based on their frequency.

2.2 On the probability of sentences

The Maximum likelihood probability of a bigram is as follow:

$$P(W_n|W_{n-1}) = \frac{C(W_{n-1}W_n)}{C(W_{n-1})}$$

$$P(W_{n-1}W_n) = \frac{C(W_{n-1}W_n)}{C(Bigrams)}$$

It is interesting to note that `C(W_n-1)` is different when using bigrams or unigrams. For example, using unigrams, we get count('not') = 7

In [41]:

```
unigram.loc[unigram['unigram']=='not','count'].values[0]
```

Out[41]:

```
7
```

But when we use bigrams, we get count('not') = 5

In [42]:

```
bigrams.loc[(bigrams['word1']=='not'),'count'].sum()
```

Out[42]:

```
5
```

The difference is caused by the fact that our bigrams do not conclude the instance of (word, $) so the last words did not get counted as the first word of a bigram.

While the difference is very small for huge data, but in our case where the corpus is small, I will take `C(W_n-1)` as count of it appearances as the first word of a bigram.

In [43]:

```
from nltk.tokenize import word_tokenize
import numpy as np

def sentence_prob(sentence):
    prob = [] #store bigram probabilities

    #word tokenization
    words =  word_tokenize(sentence)
    lower_words = [x.lower() for x in words]

    #calculate joint probability for the first two words
    logic1=((bigrams['word1']==lower_words[0])&(bigrams['word2']==lower_words[1
]))
    ct_w1_w2 = bigrams.loc[logic1,'count'].values[0]
    total_bigrams = bigrams['count'].sum()
    prob_w1_w2 = ct_w1_w2/total_bigrams
    prob.append(prob_w1_w2)

    print(f'Count of ’{lower_words[0]} {lower_words[1]}’: {ct_w1_w2}')
    print(f'Count of total bigrams: {total_bigrams}')
    print(f'P({lower_words[0]} {lower_words[1]}) = {ct_w1_w2}/{total_bigrams} =
{prob_w1_w2}')
    print('--------------------------------')

    #calculate the conditional probability of the rest of the words
    for i in range(2,len(lower_words)):
        #count of bigrams
        logic2 = ((bigrams['word1']==lower_words[i-1])&(bigrams['word2']==lower_
words[i]))
        bigram_count = bigrams.loc[logic2,'count'].values

        if bigram_count.size == 0: #if the bigram is not in the corpus
            bigram_count = 0 #count = 0
        else:
            bigram_count = bigram_count[0] #get the count

        unigram_count = bigrams.loc[bigrams['word1']==lower_words[i-1],'count'].
sum()
        prob.append(bigram_count/unigram_count) #calculate bigram probability
        sentence_prob = np.prod(prob) #calculate sentence probability

        print(f'Count of ‘{lower_words[i-1]} {lower_words[i]}‘: {bigram_count}')
        print(f'Count of ‘{lower_words[i-1]}‘: {unigram_count}')
        print(f'P({lower_words[i]}|{lower_words[i-1]}) = {bigram_count}/{unigram
_count} = {bigram_count/unigram_count}')
        print('--------------------------------')

    print(f'Probabilty of the Sentence = product of each probability above = {se
ntence_prob}')
```

1. ”It is not a good word for that"

In [44]:

```
sentence_prob('It is not a good word for that')
```

```
Count of 'it is': 3
Count of total bigrams: 547
P(it is) = 3/547 = 0.005484460694698354
--------------------------------
Count of 'is not': 2
Count of 'is': 10
P(not|is) = 2/10 = 0.2
--------------------------------
Count of 'not a': 1
Count of 'not': 5
P(a|not) = 1/5 = 0.2
--------------------------------
Count of 'a good': 1
Count of 'a': 12
P(good|a) = 1/12 = 0.08333333333333333
--------------------------------
Count of 'good word': 1
Count of 'good': 2
P(word|good) = 1/2 = 0.5
--------------------------------
Count of 'word for': 1
Count of 'word': 1
P(for|word) = 1/1 = 1.0
--------------------------------
Count of 'for that': 1
Count of 'for': 12
P(that|for) = 1/12 = 0.08333333333333333
--------------------------------
Probabilty of the Sentence = product of each probability above = 7.6
17306520414382e-07
```

2. "You must indeed go for your own good"

In [45]:

```
sentence_prob('You must indeed go for your own good')
```

```
Count of 'you must': 4
Count of total bigrams: 547
P(you must) = 4/547 = 0.007312614259597806
--------------------------------
Count of 'must indeed': 1
Count of 'must': 6
P(indeed|must) = 1/6 = 0.16666666666666666
--------------------------------
Count of 'indeed go': 1
Count of 'indeed': 2
P(go|indeed) = 1/2 = 0.5
--------------------------------
Count of 'go for': 1
Count of 'go': 4
P(for|go) = 1/4 = 0.25
--------------------------------
Count of 'for your': 1
Count of 'for': 12
P(your|for) = 1/12 = 0.08333333333333333
--------------------------------
Count of 'your own': 1
Count of 'your': 3
P(own|your) = 1/3 = 0.3333333333333333
--------------------------------
Count of 'own good': 0
Count of 'own': 2
P(good|own) = 0/2 = 0.0
--------------------------------
Probabilty of the Sentence = product of each probability above = 0.0
```

3. "How can you mistake flatter me"

In [46]:

```
sentence_prob('How can you mistake flatter me')
```

```
Count of 'how can': 3
Count of total bigrams: 547
P(how can) = 3/547 = 0.005484460694698354
--------------------------------
Count of 'can you': 2
Count of 'can': 3
P(you|can) = 2/3 = 0.6666666666666666
--------------------------------
Count of 'you mistake': 1
Count of 'you': 22
P(mistake|you) = 1/22 = 0.045454545454545456
--------------------------------
Count of 'mistake flatter': 0
Count of 'mistake': 1
P(flatter|mistake) = 0/1 = 0.0
--------------------------------
Count of 'flatter me': 1
Count of 'flatter': 1
P(me|flatter) = 1/1 = 1.0
--------------------------------
Probabilty of the Sentence = product of each probability above = 0.0
```

4. "Of them much to be for my dear"

In [47]:

```
sentence_prob('Of them much to be for my dear')
```

```
Count of 'of them': 5
Count of total bigrams: 547
P(of them) = 5/547 = 0.009140767824497258
--------------------------------
Count of 'them much': 1
Count of 'them': 6
P(much|them) = 1/6 = 0.16666666666666666
--------------------------------
Count of 'much to': 1
Count of 'much': 2
P(to|much) = 1/2 = 0.5
--------------------------------
Count of 'to be': 2
Count of 'to': 13
P(be|to) = 2/13 = 0.15384615384615385
--------------------------------
Count of 'be for': 1
Count of 'be': 8
P(for|be) = 1/8 = 0.125
--------------------------------
Count of 'for my': 2
Count of 'for': 12
P(my|for) = 2/12 = 0.16666666666666666
--------------------------------
Count of 'my dear': 6
Count of 'my': 9
P(dear|my) = 6/9 = 0.6666666666666666
--------------------------------
Probabilty of the Sentence = product of each probability above = 1.6
276295983791412e-06
```

2.3 On the interpretation of these probabilities

The zero probabilities of sentence 2 and 3 illustrate the problem of sparsity. By observing the bigram count of each component of the sentences, we can see that the bigram "own good" for sentence 2 and "mistake flatter" for sentence 3 are not in the corpus and thus have count of zero. As a result, the bigram probability and sentence probability also become zero.

We can also see that probabilities of sentence 1 and 4 are non-zero. It is normal for sentence 1 since it is grammatically correct, but abnormal for sentence 4 that consists of words in random orders. This is the problem with using a bigram model: each word is contingent on one previous word only without capturing the comprehensive context.

3. Laplace (Add One) Smoothing

Remember the Maximum Likelihood Estimation of bigrams:

$$P(W_n|W_{n-1}) = \frac{C(W_{n-1}W_n)}{C(W_{n-1})}$$

$$P(W_{n-1}W_n) = \frac{C(W_{n-1}W_n)}{C(Bigrams)}$$

Now, with Laplace add-one smoothing, the count of each bigram `P(W_{n-1} W_{n})` is added by one, and each `W_{n-1}` that appears as the first entry in a bigram is needed to increase by V, which is the unique number of unigrams.

For example, before smoothing, P(word V |word2) = 2 /sum(column of word 2):

	word 1	**word 2**	**...**	**word V-1**	**word V**
word 1	1	2	...	0	0
word 2	0	5	...	1	0
...	...	...	...	...	...
word V-1	7	0	...	0	1
word V	8	2	...	3	2

After smoothing, each cell is increased by 1, so P(word V |word2) = 2+1 /sum(column of word 2)+V:

	word 1	**word 2**	**...**	**word V-1**	**word V**
word 1	1+1	2+1	...+1	0+1	0+1
word 2	0+1	5+1	...+1	1+1	0+1
...	...+1	...+1	...+1	...+1	...+1
word V-1	7+1	0+1	...+1	0+1	1+1
word V	8+1	2+1	...+1	3+1	2+1

Therefore, the formula for the conditional probability of bigram after smoothing is:

$$P(W_n|W_{n-1}) = \frac{C(W_{n-1}W_n) + 1}{C(W_{n-1}) + V}$$

With total number of bigrams increases by V^2 (bigrams with V different first entries all increase by V), we have the joint probability:

$$P(W_{n-1}W_n) = \frac{C(W_{n-1}W_n) + 1}{C(Bigrams) + V^2}$$

Still, I will take `C(W_n-1)` as count of it appearances as the first word of a bigram.

In [56]:

```
def add_one_smoothing_prob(sentence, verbose=True):
    prob = [] #store bigram probabilities

    #word tokenization
    words =  word_tokenize(sentence)
    lower_words = [x.lower() for x in words]

    #calculate oint probability  for the first two words
    logic1= ((bigrams['word1']==lower_words[0])&(bigrams['word2']==lower_words[1
]))
    ct_w1_w2 =bigrams.loc[logic1,'count'].values[0] + 1
    total_bigrams = bigrams['count'].sum() + (len(unigram))**2
    prob_w1_w2 = ct_w1_w2/total_bigrams
    prob.append(prob_w1_w2)

    if verbose:
        print(f'Count of '{lower_words[0]} {lower_words[1]}': {ct_w1_w2}')
        print(f'Count of total bigrams: {total_bigrams}')
        print(f'Probability of P({lower_words[0]} {lower_words[1]}) = {ct_w1_w2}
/{total_bigrams} = {prob_w1_w2}')
        print('--------------------------------')

    #calculate the conditional probability of the rest of the words
    for i in range(2,len(lower_words)):
        #count of bigram
        logic2= ((bigrams['word1']==lower_words[i-1])&(bigrams['word2']==lower_w
ords[i]))
        bigram_count= bigrams.loc[logic2,'count'].values

        if bigram_count.size == 0: #if the bigram is not in the corpus
            bigram_count = 1 #count = 1 with smoothing
        else:
            bigram_count = bigram_count[0]+1 #get the count from corpus +1

        #count of unigram
        unigram_count = bigrams.loc[bigrams['word1']==lower_words[i-1],'count'].
sum() + len(unigram)
        prob.append(bigram_count/unigram_count) #calculate bigram probability
        sentence_prob = np.prod(prob) #calculate sentence probability

        if verbose:
            print(f'Count of '{lower_words[i-1]} {lower_words[i]}': {bigram_coun
t}')
            print(f'Count of '{lower_words[i-1]}': {unigram_count}')
            print(f'P({lower_words[i]}|{lower_words[i-1]}) = {bigram_count}/{uni
gram_count} = {bigram_count/unigram_count}')
            print('--------------------------------')

    print(f'Probabilty of "{sentence}" = {sentence_prob}')
```

Now let's observe the new probabilities of the four sentences.

In [57]:

```
add_one_smoothing_prob('It is not a good word for that', False) #sentence 1
```

```
Probabilty of "It is not a good word for that" = 2.4778831051678218e
-17
```

In [58]:

```
add_one_smoothing_prob('You must indeed go for your own good', False) #sentence
 2
```

```
Probabilty of "You must indeed go for your own good" = 1.08691677051
42373e-17
```

In [59]:

```
add_one_smoothing_prob('How can you mistake flatter me',False) #sentence 3
```

```
Probabilty of "How can you mistake flatter me" = 1.977589590025535e-
13
```

In [60]:

```
add_one_smoothing_prob('Of them much to be for my dear',False) #sentence 4
```

```
Probabilty of "Of them much to be for my dear" = 1.8898708715474837e
-16
```

By redistributing counts from seen to unseen bigrams, the original zero-probability sentence 2 and 3 now have small positive probabilities, and sentence 1 and 4 with previously non-zero probabilities now have smaller probabilities. The sparsity problem is solved.

However, the incapability of accurately capturing contexts associated with the bigram model is not resolved as the probability of sentence 4 is still non-zero.