NLP 1

Emmanuel Adebayo

January 2024

1 Problem 1

1. (a) $\begin{bmatrix} 2 & 7 & 7 \\ 4 & 3 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} (2*0)+(7*1)+(7*1) & (2*2)+(7*0)+(7*1) \\ (4*0)+(3*1)+(1*1) & (4*2)+(3*)+(1*1) \end{bmatrix}$
$= \begin{bmatrix} 14 & 11 \\ 4 & 9 \end{bmatrix}$

   (b) $\begin{bmatrix} 0 & 6 \\ 9 & 2 \\ 2 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 7 & 3 & 1 \end{bmatrix} = \begin{bmatrix} (0*1)+(6*7) & (0*2)+)+(6*3) & (0*3)+(6*1) \\ (9*1)+(2*7) & (9*2)+(2*2) & (9*3)+(2*1) \\ (2*1)+(1*7) & (2*2)+(1*3) & (2*3)+(1*1) \end{bmatrix}$
$= \begin{bmatrix} 42 & 18 & 6 \\ 23 & 24 & 29 \\ 9 & 7 & 7 \end{bmatrix}$

   (c) $\begin{bmatrix} 2 \\ 0 \\ 2 \end{bmatrix} * \begin{bmatrix} 4 & 1 & 4 \\ 4 & 4 & 0 \\ 1 & 0 & 3 \end{bmatrix} = \text{invalid}$

   (d) $\begin{bmatrix} 3 \\ 1 \\ 0 \end{bmatrix} * \begin{bmatrix} 1 & 4 & 5 \end{bmatrix} = \begin{bmatrix} 3*1 & 3*4 & 3*5 \\ 1*1 & 1*4 & 1*5 \\ 0*0 & 0*4 & 0*5 \end{bmatrix} = \begin{bmatrix} 3 & 12 & 15 \\ 1 & 4 & 5 \\ 0 & 0 & 0 \end{bmatrix}$

   (e) $\begin{bmatrix} 1 & 7 \\ 4 & 0 \end{bmatrix} * \begin{bmatrix} 6 & 8 \end{bmatrix} = \text{invalid}$

2 Problem 2

1. (a) `^(?:[a-z]+)(?:  [a-z]+)*$`

   (b) `^(?:[A-Z][a-zA-Z0-9]*[.,;:’"()?!]*(?:\s|$))+`

   (c) `(ice)(?!-|[A-Za-z])`

   (d) `^[a-z]*b$`

   (e) `^[0-9]+\s*[A-Za-z]+$`

3 Problem 3

(a) i. I picked Yoruba

ii. The morphological complexity in Yoruba can make tokenization and part-of-speech tagging more challenging compared to languages with simpler structures. Ambiguities in word boundaries and morphological variations pose obstacles for accurate analysis and interpretation by NLP algorithms.

Different tones can change the meaning of a word, and accurately capturing these variations in speech requires sophisticated models that can distinguish tonal nuances. Without proper handling of tonal information, ASR systems may misinterpret words, leading to errors in transcriptions.

(b) I chose the The AG's news topic classification dataset is constructed by choosing 4 largest classes from the original corpus.
https://www.kaggle.com/datasets/amananandrai/ag-news-classification-dataset?resource=download
Motivation: The dataset is provided by the academic comunity for research purposes in data mining (clustering, classification, etc), information retrieval (ranking, search, etc), xml, data compression, data streaming, and any other non-commercial activity. It was provided by ComeToMyHead, which is an academic news search engine which has been running since July, 2004
Situation: The AG's news topic classification dataset is constructed by choosing 4 largest classes from the original corpus
Collection process: The AG's news topic classification dataset is constructed by choosing 4 largest classes from the original corpus. The AG's news topic classification dataset is constructed by Xiang Zhang (xiang.zhang@nyu.edu). They used collection of more than 1 million news articles
Annotation Process: It contains the following features – title, description, and class index (label). It was annotated by Xiang Zhang (xiang.zhang@nyu.edu). Each class contains 30,000 training samples and 1,900 testing samples. The total number of training samples is 120,000 and testing 7,600. The files train.csv and test.csv contain all the training samples as comma-sparated values. There are 3 columns in them, corresponding to class index (1 to 4), title and description. The title and description are escaped using double quotes ("), and any internal double quote is escaped by 2 double quotes (""). New lines are escaped by a backslash followed with an "n" character.

4 problem 4

1. (a) The total number of sentences are: 24580

```
The total number of sentences are:  24580
```

```python
def split_using_sent_tokenize_then_use_python_split_to_split_the_sentences_and_nltk_tokeni
    #holds the total number of sentences
    self.num_of_sentences = 0
    #holds the total number of token using python split
    self.num_of_words_with_python_split = 0
    #holds the total number of token using nltk word tokenize split
    self.num_of_words_with_nltk_tokenize = 0

    #loop through the data.txt file
    with open(self.output_path, 'r') as output_file:
      for line in output_file:
        #tokenize the sentence
        sentences = sent_tokenize(line)
        #check if token is empty
        #if empty, do not bother
        if len(sentences) ==0:
          continue
        if '' in sentences:
          continue
        if "" in sentences:
          continue

        if sentences:
          #add number of sentences
          self.num_of_sentences += len(sentences)
          #split the sentences using python split to count
          self.num_of_words_with_python_split += self.split_method_to_count(sentences)
          #using the the nltk to count
          self.num_of_words_with_nltk_tokenize += self.tokenize_nltk_to_count(sentences)


    print("The total number of sentences are: ", self.num_of_sentences)

    print("The total number of tokens using python split: ", self.num_of_words_with_python

    print("The total number of tokens using nltk tokenize: ", self.num_of_words_with_nltk_
```

(b) The total number of tokens using python split: 273355

```
The total number of tokens using python split:  273355
```

```
#an helper method to split
#the line into sentences
def split_method_to_count(self, sentence):
  num_of_words = 0
  #loop through sentence and use python split to turn into array
  #get the length of the array. which is the number
  #of the words
  for sent in sentence:
    words_split = sent.split(' ')
    num_of_words += len(words_split)

  #return the total number of tokens
  return num_of_words
```

(c) The total number of tokens using nltk tokenize: 321232

```
the total number of tokens in lowercased words using nltk package 321151
```

```
#an help method to split
#uses nltk word tokenize
def tokenize_nltk_to_count(self, sentence):
  num_of_words = 0
  #loop through the sentence
  #for each sentence
  #tokenize and get the length
  for sent in sentence:
    sent_tokenized = word_tokenize(sent)
    num_of_words += len(sent_tokenized)

  #return the total number of tokens
  return num_of_words
```

(d) the total number of tokens in lowercased words using nltk package is 321151
the total number of types in lowercased words nltk package is 8786

```
the total number of tokens in lowercased words using nltk package 321151
the total number of types in lowercased words nltk package 8786
```

```python
#count token types from the lowercased.txt
def count_tokens_and_types_after_lowercasing(self):
  #set to count token
  #it avoids duplicates
  types = set()
  #holds total token
  total_token = 0
  #to track the number of duplicates
  duplicates = 0
  #a dictionary to count token frequencies
  lower_dict = {}
  with open(self.lowercased_file_path_of_data_txt, 'r') as low_path_file:
    for line in low_path_file:
      #tokenize the line into sentences
      line_sent_tokenize = sent_tokenize(line)
      # print(line_sent_tokenize, len(line_sent_tokenize))

      if len(line_sent_tokenize) == 0:
        continue

      #loop through the tokenization i.e each sentences
      #loop through each tokenize sentences
      for sents in line_sent_tokenize:
        sents_to_word_tokenize = word_tokenize(sents)
        # loop through each tokens
        for tk in sents_to_word_tokenize:
          # add token to the set
          types.add(tk)
          #count duplicates
          if tk in types:
            duplicates += 1
          #track the frequency
          #with the lower_dict dictionary
          if tk in lower_dict:
            lower_dict[tk] += 1
          else:
```

the total number of types in lowercased words 8786
the total number of types in lowercased words using dictionary 8786
found this much duplicates 321151

(e) I suspect the answers are different because different tokenization methods such as python's split and ntlk word tokenize use different mechanism in tokenization.
Also, after lowercasing all the word, uppercase words and lowercase words become one, hereby shortening the amount of unique types

(f) the most freqent word type in lowercased version is: i
the most freqent word in data.txt (pre lowercasing) using nltk package is: .

```
the most freqent word type in lowercased version is:  i
the fifth most freqent word lowercased version is:  you
the most freqent word in data.txt (pre lowercasing) using nltk package is:  .
the fifth most freqent word in data.txt (pre lowercasing) using nltk package is:  you
```

```python
#from the not pre lowercased file
def get_words_types_and_frequency(self):
  #make a dictionary
  word_dict = {}
  with open(self.output_path, 'r') as output_file:
    for line in output_file:
      tokenized_word = word_tokenize(line)
      for word in tokenized_word:
        if word in word_dict:
          word_dict[word] += 1
        else:
          word_dict[word] = 1


  sorted_dict = dict(sorted(word_dict.items(), key=lambda item: item[1], reverse =True))

  word_type = next(iter(sorted_dict))
  print("the most freqent word in data.txt (pre lowercasing) using nltk package is: ",
        word_type)

  counter = 0
  for key, value in sorted_dict.items():
    counter += 1

    if counter > 4:
      print("the fifth most freqent word in data.txt (pre lowercasing) using nltk package
            key)
      break

  ranked_words = list(word_dict.keys())
  frequencies = list(word_dict.values())


  # Plotting the graph
  plt.figure(figsize=(10, 6))
```

(g) the fifth most freqent word lowercased version is: you
the fifth most freqent word in data.txt (pre lowercasing) using nltk package is: you

```
the most freqent word type in lowercased version is:  i
the fifth most freqent word lowercased version is:  you
the most freqent word in data.txt (pre lowercasing) using nltk package is:  .
the fifth most freqent word in data.txt (pre lowercasing) using nltk package is:  you
```

```python
#from the not pre lowercased file
def get_words_types_and_frequency(self):
  #make a dictionary
  word_dict = {}
  with open(self.output_path, 'r') as output_file:
    for line in output_file:
      tokenized_word = word_tokenize(line)
      for word in tokenized_word:
        if word in word_dict:
          word_dict[word] += 1
        else:
          word_dict[word] = 1


  sorted_dict = dict(sorted(word_dict.items(), key=lambda item: item[1], reverse =True))

  word_type = next(iter(sorted_dict))
  print("the most freqent word in data.txt (pre lowercasing) using nltk package is: ",
        word_type)

  counter = 0
  for key, value in sorted_dict.items():
    counter += 1

    if counter > 4:
      print("the fifth most freqent word in data.txt (pre lowercasing) using nltk package
            key)
      break

  ranked_words = list(word_dict.keys())
  frequencies = list(word_dict.values())


  # Plotting the graph
  plt.figure(figsize=(10, 6))
```

(h) The graph below evidently follows zipf's law. common words occur frequently while uncommon words have less frequency