task2_31187366

September 16, 2020

1 FIT5196 Assessment 1

Student Name: PRASHANT JAJORIA

Student ID: 31187366 Date: 13/09/2020

Environment: Python 3.7.4 and Anaconda 4.8.4 (64-bit)

Libraries used: * pandas - for reading excel into Dataframe. And dataframe manuplation. * langid - for finding English tweets, included in Anaconda Python 3.7.4 * nltk - (Natural Language Toolkit, included in Anaconda Python 3.7.4)

- nltk.collocations - for finding bigrams, included in Anaconda Python 3.7.4
- nltk.tokenize - for Regex tokenization and Multiple word expression tokenization, included in Anaconda Python 3.7.4
- nltk.probability - for calculating frequency distribution of tokens and finding most common unigrams and bigrams, included in Anaconda Python 3.7.4
- sklearn.feature_extraction.text - for calculating count vector using the `CountVectorizer`
- nltk.util - for calculating the bigrams using the `ngrams` function
- nltk.stem - for stemming of tokens using the PorterStemmer

1.1 1. Introduction

The main goal of this assignment is to convert exracted data to into a proper format, which can be used by down stream data analysis algorithms. An excel file with 81 sheets is provided with Tweets about Covid-19 for different dates. The main objective in this task is to tokenize the data and get the data in format that can be used by downstream data analyais algorithms .

Following are the requirement of the task: 1. Only English tweets should be processed for tokenization. 2. Generate a vocabulary file with the tokens sorted alphabitacilly. 3. For each day, to calculate the top 100 unigrams and generate a top 100 unigrams text file 4. For each day, to calculate the top 100 bigrams and generate a top 100 bigrams text file 5. Generate a sparse representation (i.e., doc-term matrix) of the excel file with provided format.

A step by step explanation for completing the requirements will be explained in the following code cells.

1.2 2. Import Libraries

[498]:
```
import pandas as pd
import langid
import nltk

from nltk.tokenize import RegexpTokenizer
from itertools import chain
from nltk.probability import *
from sklearn.feature_extraction.text import CountVectorizer
from nltk.util import ngrams
from nltk.stem import PorterStemmer
from nltk.tokenize import MWETokenizer
```

1.3 3. Reading the Excel file

Reading the Excel file using the Pandas `ExcelFile` function.

[499]:
```
# reading my excel file
file = pd.ExcelFile("31187366.xlsx")
```

Storing the name of each sheet of the excel file

[500]:
```
# getting the sheet names of excel file
sheets = file.sheet_names
```

Declaring variable that will store the concatenated Tweets text for each date in a dictionary

[501]:
```
# declaring dictionary to store the tweets
dict_data = {}
```

1.4 4. Parsing, cleaning and filtering Non English tweets

- Reading each sheet of Excel file using the `parse` method and storing it as a Dataframe.
- Removing all the NA columns and rows using the `dropna` with `how` = 1 for columns , and `how` = 0 for rows.
- Resetting the column name for each data frame conditionally using the `reset_index` function. And dropping the old column names from the dataframe.
- Using `apply` to use the `lambda` function along with `langid.calssify` fucntion on `Text` column to have only English Tweets in the data frame.
- Getting the all the values of the Text column and joining using the new line character `\n` using the `join` function.
- Finally storing the values of each date in a Dictionary with key being the Date and value is the concatenated string having all the tweet text of that date.

[ ]:
```
# looping each sheet
for sheet in sheets:
```

```
    #print(sheet)

    # get each sheet
    df = file.parse(sheet_name = sheet)

    # remove all rows and columns with NA values
    df = df.dropna(0,how="all")
    df = df.dropna(1,how="all")


    if(df.columns[0] != "text" and df.columns[1] != "id" and df.columns[2] !=␣
→"created_at" ):
        # changing the column names
        df.columns = df.iloc[0]

        # removing old colmuns
        df.drop(df.index[0],inplace = True)

    df.reset_index(drop=True, inplace=True)

    # classify the tweets and store the result TRUE/FALSE in an column "result"
    df["result"] = df['text'].apply(lambda x : langid.classify(str(x)))

    # remove the non-english tweets
    df = df[df["result"].apply(lambda x : x[0] == "en")]

    # get the values column
    tweet_values = df.text.values

    # store in list if the text is string
    tweets = [ tweet for tweet in tweet_values if isinstance(tweet, str)]

    # join all the tweets and store as dictionary
    dict_data[sheet] = '\n'.join(tweets)
```

1.5 5. Tokenization using Regexp Tokenizer

- Using the Regexp Tekenizer form nltk.tokenize pacakage with the provided Regular Expression `[a-zA-Z]+(?:[-'][a-zA-Z]+)?` to tokenize each concatenated string of a date.
- Following is the explanation of each part of the Regular Expression:
  - `[a-zA-Z]` : Accepts atleast one or more upper or lower case or mix of both characters.
  - `(?:[-'][a-zA-Z]+)?` : This is an optional non capturing group. It captures words that start with either `-` or `'`, followed by atleast one or more upper or lower case or mix of both case characters.
- The result of Tokenization is converted to lower case and stored in a Dictionary with key as the Date and value as the list of Tokens of that date.

[503]:
```python
# tokenizing the tweets
dict_unigram_token = {}

# regex tokenizer
tokenizer = RegexpTokenizer(r"[a-zA-Z]+(?:[-'][a-zA-Z]+)?")

# do the tokenization for each date
for date, tweets in dict_data.items():

    # lowercase the string
    tweets = tweets.lower()

    # store the unigram tokens in a dictionary
    dict_unigram_token[date] = tokenizer.tokenize(tweets)
```

1.6 6. Calculating the top 100 frequent Bigarms for each Day

After tokenization we calculate the Bigrams for each day first. This step need tobe carried out before Stemming of tokens, as after stemming the words will lose it meaning due to characters being taken off, and we are only left with the root word.

Using the `ngrams` function from NLTK pacjakage allows us to calculate the Bigrams by specifying the parameter size as `n=2`.

Using the result then we do a frequency distribution to get the Frequency of each Bigram in the documents.

After this the using the `most_common` function we get the top 100 Bigrams from all the sheets.

[504]:
```python
# CALCULATING 100 FREQUENT BIGRAMS FOR A DAY

dict_top_100_bigrams = {}

for date, list_unigram_tokens in dict_unigram_token.items():

    bigrams = ngrams(list_unigram_tokens, n = 2)
    fdbigram = FreqDist(bigrams)

    dict_top_100_bigrams[date] = fdbigram.most_common(100)
```

1.7 7. Write the top 100 frequent Bigrams to .txt file

After having the top 100 Bigrams, we write those to the `31187366_100bi.txt` file using the format `Bigram:Count`.

[505]:
```python
# generate the top 100 BIGRAM file
out_file = open("./final_op/31187366_100bi.txt", 'w')

for k,v in dict_top_100_bigrams.items():
```

```
        out_file.write('{}:{}'.format(k,v) + '\n')

out_file.close()
```

1.8 8. Removing tokens with length less than 3 from the vocab

As one of the requiement of the task we do not process tokens of size less than 3. So removing those from the vocab.

Using List comprehension to make a new list of Tokens.

[506]:
```
# removing tokens with length less than 3
for date, tokens in dict_unigram_token.items():
    dict_unigram_token[date] = [ word for word in filter(lambda x : len(x) >=␣
 ↪3,tokens)]
```

1.9 9. Reading the Stopwords file

When we see the tokens, we find that most of the tokens are common English grammar words that are of no meaning independently. These are Stop words and are not imortant for our analysis and are removed from the vocublary. These stopwords are called **Context Independent Stopwords.**

Reading the given `stopwords_en.txt` file to get the stopwords.

[507]:
```
# create a list of stop words
stopwords = []

#open the stopwords file
with open('./stopwords_en.txt') as f:

    # read the stop word line by line
    stopwords = f.read().splitlines()
```

1.10 10. Removing Context Independent Stop words from the vocab

Using the given stopwords file we check the membershipof each token in the Stop words list. And we make vocab using List comprehension after removing Stopwords

[508]:
```
# CONTEXT INDEPENDENT STOP WORDS REMOVAL

# make a new dictionary after removing the stop words
dict_unigram_token_no_stop_words = {}

# convert to set
stopwords_set = set(stopwords)

# for each sheet, reomove the stop words
for sheet in sheets:
```

```
    # store it in a new dictionary
    dict_unigram_token_no_stop_words[sheet] = [word for word in␣
↪dict_unigram_token[sheet] if word not in stopwords_set]
```

1.11 11. Calculating the Frequency distribution of Tokens

Calculating the Frequency Distribution of all the words after removing the Stopwords. Using `FreqDist()` to do this frequency Ditribution.

[509]:
```
# allwords after reomving the words with less than 3
list_no_stop = list(chain.from_iterable([set(value) for value in␣
↪dict_unigram_token_no_stop_words.values()]))
fd_no_stop = FreqDist(list_no_stop)
```

1.11.1 Extracting words that appear in less than 5 documents

Using the previous Frequency Distribution object obtained, we make a list of words that appear in less than 5 documents.

[510]:
```
# get the words which appera in less than 5 documents
doc_ref_less_5 = set([w[0] for w in fd_no_stop.items() if w[1] < 5])
#doc_ref_less_5
```

1.11.2 Extracting words that appear in more than 60 documents

Using the previous Frequency Distribution object obtained, we make a list of words that appear in more than 60 documents.

[511]:
```
# get the words which appera in more than 60 documents
doc_ref_more_60 = set([w[0] for w in fd_no_stop.items() if w[1] > 60])
#doc_ref_more_60
```

1.12 12. Removing Context Dependent Stopwords

Using the preiously created List of Context independent tokens of frequency less than 5 and frequency more than 60, we check for Membership f each token in both the list, and make a new List using List comprehension

[512]:
```
# CONTEXT DEPENDENT STOP WORDS REMOVAL

# new dictionary after removing the less common words
dict_unigram_token_less_freq_rem = {}

# for each sheet, reomove the words which appears in less than 5 documents
for sheet in sheets:

    # store it in a new dictionary
```

```
    dict_unigram_token_less_freq_rem[sheet] = [w for w in␣
↪dict_unigram_token_no_stop_words[sheet] if w not in doc_ref_less_5 and w not␣
↪in doc_ref_more_60]
```

1.13 13. Stemming using Porter stemmer

Using **PorterStemmer** defined in the NLTK.stem package, we stem all the tokens and get a new vocab. We can do the Stemming now as all the Context dependent and context Independent tokensa are removed. And also all the Bigrams for each day are already calculated.

Using the `stem` function we carry out stemming of each token.

[513]:
```
#STEMMING
stemmer = PorterStemmer()

dict_stemmed_unigram_tokens = {}

for k,v in dict_unigram_token_less_freq_rem.items():
    dict_stemmed_unigram_tokens[k] = [ stemmer.stem(w) for w in v]
```

1.13.1 List of all Stemmed tokens

We use the `from_iterable()` method from `chain` pacakge to create a List of all the Stemmed tokens from every file.

Using a `set()` to keep only one instance of the token for a day in the final tokens list.

[514]:
```
tokens_list = list(chain.from_iterable([set(value) for value in␣
↪dict_stemmed_unigram_tokens.values()]))
```

1.14 14. Calculating the top 100 frequent unigrams for each day

As the tokens are stemmed, the vocabulary is reduced keeping only the root words and other important words that are in the documents.

We calculate the Frequency distribution of tokens of each day after stemming. This gets the frequency of tokens of each day.

Again we use the `most_common()` function to get the top 100 Unigrams of each day.

[515]:
```
# calculating the top 100 unigrams for each day
dict_100_unigram_each_day = {}

# new dict ot store the top 100 unigrams token
dict_unigrams = {}

for sheet in sheets:
    freqDist = FreqDist(dict_stemmed_unigram_tokens[sheet])
    dict_100_unigram_each_day[sheet] = freqDist.most_common(100)
```

1.15 15. Write the top 100 frequent Unigrams to .txt file

Writing the top 100 unigarms to the `31187366_100uni.txt` file. We use the format of `Unigram:Count` for writitng to the file.

[516]:
```
# generate the top 100 UNIGRAM file
out_file = open("./final_op/31187366_100uni.txt", 'w')

for k,v in dict_100_unigram_each_day.items():
        out_file.write('{}:{}'.format(k,v) + '\n')

out_file.close()
```

1.16 16. Calculating 200 meaningful Bigrams

We can proceed with the task of finding the most meaningful Bigrams from all the tokens, as the Context Dependent and Context Independentstop words are already removed.

We use the `BigramCollocationFinder` defined in the `nltk.collocations` package to get the Bigrams.

Then using the result we got from the `BigramCollocationFinder` we use `PMI` as an association measure to know how important the Bigrams are in the documents.

We extract to top 200 best Bigrams using the `nbest` function.

[522]:
```
# CALCULATING 200 MEANINGFUL BIGRAMS FROM ALL THE DAYS

bigram_measures = nltk.collocations.BigramAssocMeasures()

finder = nltk.collocations.BigramCollocationFinder.from_words(fd_no_stop)

top_200_bigrams = finder.nbest(bigram_measures.pmi, 200)

#top_200_bigrams
```

1.17 Retokenize using MWETokenizer

Aa we have the Bigrams that are meaningful, we now want to inlcude them in our vocublary. For theis we have to retokenize the vocab.

We now use the **Multiple word expression** tokenizer as we have to inlcude the Bigrams.

[523]:
```
mwetokenizer = MWETokenizer(top_200_bigrams)

dict_tokens =  dict((date, mwetokenizer.tokenize(unigrams_list)) for␣
↪date,unigrams_list in dict_stemmed_unigram_tokens.items())

#print(dict_tokens)
```

```
#print(dict_tokens.values())

all_tokens_colloc = list(chain.from_iterable(dict_tokens.values()))

list_colloc_voc = list(set(all_tokens_colloc))

#print(list_colloc_voc)
```

1.18 17. Generate Sparse representation using CountVectorizer

After tokenization we can understand that most of the tokens will not appear in all the files. So keeping a count of these tokens which have a count of 0 will add to the memory and processing overhead. So the matrix of document v/s count will be sparse as for most of the document the value will be zero.

To solve this probem, we maintain a feature vector. Applying `fit_transform` to the new vocab.

`fit_transform` does two things: First, it fits the model and learns the vocabulary; second it transforms the text data into feature vectors.

[519]:
```
# COUNT VECTOR
vectorizer = CountVectorizer(analyzer = "word")

data_features = vectorizer.fit_transform(list([' '.join(value) for value in␣
 ↪dict_tokens.values()]))

vocab = vectorizer.get_feature_names()
```

1.19 18. Generate Vocabulary file

We maintain an dictionary mapping of each Token and assign an index to it. This index and token is sored in the vocab file.

The format of vocab file is : `token:index`

Finally writing the tokens to a vocab file `31187366_vocab.txt` file.

[520]:
```
# generate the VOCAB file
out_file = open("./final_op/31187366_vocab.txt", 'w')

i = 0

tokens_list = list(sorted(list(set(vocab))))

# get the single token
for token in tokens_list:

    # give each token an index
    vocab_dict[token] = i
```

```
    out_file.write('{}:{}'.format(token,i) + '\n')

    i = i + 1

out_file.close()
```

1.20 18. Generate Doc-Term Matrix file

Given the vocabulary, each document can be represented as a sequence of integers that correspond to the tokens, or in the following sparse form:

    word_index:word count

We keep count of only those tokens that have `count > 0` to solve the problem of sparse matrix.

The `data_features` returns a matrix of count of Documents v/s Token. Each row is count of the token for a single token.

So we can bind the together for each doument using the `zip` function.

Finlly the result is written to `31187366_countVec.txt` file.

[521]:
```
# write to countVec file
# open a file to write
out_file = open("./final_op/31187366_countVec.txt", 'w')

i=0
dict_count_token_each_day = {}

while i < len(sheets):
    count_token_each_doc = data_features.toarray()[i]

    out_file.write(sheets[i] + ",")
    for word, count in zip(vocab, count_token_each_doc):

        if count > 0:
            #print (vocab_dict[word], ":", count)
            out_file.write("{}:{},".format(vocab_dict[word],count))
    out_file.write('\n')
    i = i+1

# close the file
out_file.close()
```

1.21 Summary

This task help build up the knowledge of processing semi structured text, and converting the text to suitable format useful for downstream text analysis algorithms. The main outcomes achieved are as follows:

- **Reading Excel file and parsing various sheets** : Using the pandas ReadExcel function we could read the given excel file and work on different sheets using the parse() function.
- **DataFrame manipulation** : Storing the data in Pandas dataframe and manipulating it to get the text was important to get started with the task.
- **Tokenization** - Converting each file text to String and tokenizing the words was important to get the Unigrams, Bigrams and Collocations from the data.
- **Stemming** - Using the PorterStemmer we could see the effect of Stemming on the tokens. The conversion of words to their root form helped shorten the vocab.
- **Generating Count Vector** - It is imortant to store the count of each token for analysis. As storing the count of each token fpr each file creates a Sparse Marix, we use count vetor form to save the count.
- **Storing Data to specific format** - The process of storing the unigram and bigrams to external file helped to learn the storage of data to different format.
- **Manipulating Python Data structures**: For successful completion of this task knowledge of manipulating basic Data Structures like `list`, `tuple` and `dictionary` was important. Using dictionary functions like `items` and `keys` was helpfult for iteration. Also the `in` operator was needed for various condition check.
- **Remove Non-English Tweets** : Using `langid` package it was possible to check if the text of Tweet was in English or not. This was achieved by using the `classify` function.

1.22 References

- nltk.collocations — NLTK 3.5 documentation. (2020). Retrieved 16 September 2020, from https://www.nltk.org/_modules/nltk/collocations.html
- langid. (2020). Retrieved 15 September 2020, from https://pypi.org/project/langid/1.1dev/
- 10 Minutes to pandas — pandas 0.22.0 documentation. (2020). Retrieved 16 September 2020, from https://pandas.pydata.org/pandas-docs/version/0.22.0/10min.html
- Porter Stemming Algorithm. (2020). Retrieved 16 September 2020, from https://tartarus.org/martin/PorterStemmer/
- Stemming. (2020). Retrieved 16 September 2020, from https://en.wikipedia.org/wiki/Stemming