Lab 3: Search Terms with Pandas Pt. 2

Introduction:

This notebook is identical to my other notebook **Lab 3: Search Terms with Pandas** in everyway except for the data set that is being used. This is so that predictions can be made based on the outcomes from the previous notebook, and then those predications can then be tested and a conclusion can be drawn from the comparison of the predications to the actual results.

Author: Nigel Nelson

Filter: Lists of Lists/Tuples

The cell below is a method that takes in a list of lists/tuples, and returns a list of single objects from a given index in those sub-lists

In [1]:
```
def filter_rough_data(rough_data, index):
    data = []
    for x in rough_data:
        data.append(x[index])
    return data
```

Filter: Remove Web-Spaces

The cell below contains a method that receives a list that may or may not contain web-spaces "%20" and returns a list where these web-spaces are replaced by traditional spaces " ".

In [2]:
```
def remove_web_spaces(list):
    clean_tokens = []
    for token in list:
            clean_tokens.append(token.replace('%20', ' '))
    return clean_tokens
```

Filter: Remove Spaces

The cell below contains a method that recieves a list of search queries where a single token may be comprised of multiple words seperated by spaces, and returns a list where each entry contains no spaces and only a single word for each token.

In [3]:
```
def create_single_tokens(data):
    single_tokens = []
    for token in data:
        words = token.split()
        for single_word in words:
            single_tokens.append(single_word)
    return single_tokens
```

Create: Spell Check Dictionary

The cell below contains a method that takes a list of words that may be mispelled, and uses the **spellchecker** library to create a dictionary where the key is a word from the list that is mispelled, and the value is the correct spelling of the word.

*note: The below code has reduced spell checking accuracy as the distance used by the **SpellChecker** has been reduced to 1 to reduce run-time. This disctance can be increased to 2 to increase the spell checking accuracy but this will come at a cost to run-time.*

In [4]:
```
from spellchecker import SpellChecker

def spell_check_dict(data):
    spell = SpellChecker(distance=1)
    misspelled = spell.unknown(data)
    corrective_spelling_dict = {}
    for word in misspelled:
        corrective_spelling_dict[word] = spell.correction(word)
    return corrective_spelling_dict
```

Create: Spell Checked Tokens

The cell below contains a method that uses a spell check dictionary (*key = mispelled word, value = correctly spelled word*) and a given word, and returns the correct spelling of that word according to the provided spell check dictionary.

In [5]:
```
def spell_check(dict, word):
    correct_word = word
    if word in dict:
        correct_word = dict[word]
    return correct_word
```

Create: Frequency Dictionary

The cell below contains a method that takes a list of words, and adds it to a dictionary where the key is the word, and the value is the number of times that the given word appeared in the list.

In [6]:
```
def frequency_dict(list):
    dict = {}
    for x in list:
        if dict.get(x) == None:
            dict[x] = 1
        else:
            dict[x] += 1
    return dict
```

Sort: Frequency Dictionary

The cell below contains a method that takes a dictionary where the values are represented by a number, and returns a sorted dictionary of decending values.

In [7]:
```
def sort_dict(dict):
    temp_list = []
    for key, value in dict.items():
        temp_list.append([key, value])
    temp_list.sort(key = lambda x: x[1], reverse = True)
    decending_dict = {}
    for x in temp_list:
        decending_dict[x[0]] = x[1]
    return decending_dict
```

Data Description:

The data set that these methods are being ran on is from Tom Davidson's (t-davidson) GitHub repository *hate-speech-and-offensive-language* (https://github.com/t-davidson/hate-speech-and-offensive-language) where he supplies a .csv file of tweets that have been flagged as hate speech. In addition to the tweet, the csv file contains counts of the number of users who coded each tweet, the number of users who judged the tweet to be hate speech, the number of users who judged the tweet to be offensive, the number of users who judged the tweet to be neither offensive nor non-offensive, and a categorization of whether the tweet was hate speech, offensive, or neither. This csv file is 2.43 MB in size.

In [8]:
```
import csv

with open('labeled_data.csv', 'r') as f:
    reader = csv.reader(f)
    rough_data = list(reader)
```

Executing Filters on Data Set

The cell below contains calls to the methods above that results tokens to be:

1. Filtered to only contain search queries and not abitrary characters that are in the tuples held in the tokens.
2. Filtered to remove web-spaces "%20" from the tokens in the data.
3. Filtered to consolodate the tokens to only single tokens with no spaces

In [9]:
```
#1
print('Benchmarking for step #1:')
%time filtered_data = filter_rough_data(rough_data, 6)

#2
print('\nBenchmarking for step #2:')
%time clean_data = remove_web_spaces(filtered_data)

#3
print('\nBenchmarking for step #3:')
%time single_tokens = create_single_tokens(clean_data)
```

```
Benchmarking for step #1:
CPU times: user 1.8 ms, sys: 0 ns, total: 1.8 ms
Wall time: 1.81 ms

Benchmarking for step #2:
CPU times: user 4.55 ms, sys: 0 ns, total: 4.55 ms
Wall time: 4.55 ms

Benchmarking for step #3:
CPU times: user 37.8 ms, sys: 4.15 ms, total: 42 ms
Wall time: 41.2 ms
```

True Size of Data Set

The below cell is reponsible for displaying the number of bytes used to store the **single_tokens** list, which is just the list of the individual words in tweets and sheds the unnecessary counts that are associated with each tweet. This will be used in order to make benchmark predictions for all non-trivial cells.

In [10]:
```
import sys

sys.getsizeof(single_tokens)
```

Out[10]:
```
3012912
```

Use of Pandas

One of the primary goals of this notebook is to leverage pandas, which is a powerful open source data analysis and manipulation tool. As such, the pandas libray is imported in the below cell so it can be used for the remainder of the notebook.

In [11]:
```
import pandas as pd
```

Creating a DataFrame

The below cell take the **single_tokens** list and creates a single columned data frame **df**

Predictions:

Understanding that this data set of single tokens is about 1/4th the size of the single tokens from Direct Supply's DSSI eProcurement system's search queries which has already been benchmarked. I would hypothesize that the below cell takes about **30 ms** to execute.

In [12]:
```
%%time

df = pd.DataFrame({'single_tokens' : single_tokens})
```

```
CPU times: user 27.2 ms, sys: 486 µs, total: 27.7 ms
Wall time: 29.2 ms
```

Converting to Lowercase

The below cell adds a new column to the data frame **df** called **lower_case** which uses the tokens from **single_tokens** and applies a lambda expression which ensure each letter is represented in its lowercase form.

Predictions:

Understanding that the original data set of single tokens is about 1/4th the size of the single tokens from Direct Supply's DSSI eProcurement system's search queries which has already been benchmarked. I would hypothesize that the below cell takes about **65 ms** to execute.

In [13]:
```
%%time

df['lower_case'] = df['single_tokens'].apply(lambda s: s.lower())

df.head()
```

```
CPU times: user 62.1 ms, sys: 7.75 ms, total: 69.9 ms
Wall time: 70 ms
```

Out[13]:

	single_tokens	lower_case
0	tweet	tweet
1	!!!	!!!
2	RT	rt
3	@mayasolovely:	@mayasolovely:
4	As	as

Removing Non-Letters

The below cell is repsonsible for creating a new column **letters_only** which uses the tokens from **lower_case** and strips all characters that are non-letters.

Predictions:

Understanding that the original data set of single tokens is about 1/4th the size of the single tokens from Direct Supply's DSSI eProcurement system's search queries which has already been benchmarked. I would hypothesize that the below cell takes about **345 ms** to execute.

In [14]:
```
%%time

import re

df['letters_only'] = df['lower_case'].apply(lambda s: re.sub(r'[^A-Za-z]','',s
))

df.head()
```

```
CPU times: user 333 ms, sys: 3.7 ms, total: 337 ms
Wall time: 336 ms
```

Out[14]:

	single_tokens	lower_case	letters_only
0	tweet	tweet	tweet
1	!!!	!!!	
2	RT	rt	rt
3	@mayasolovely:	@mayasolovely:	mayasolovely
4	As	as	as

Replacing Empty Strings

The below cell is responsible for creating a new column **none_values** which uses the tokens from **letters_only** and replaces all empty strings with a 'None' value so that it is easy to identify rows that contain empty values.

Predictions:

Understanding that the original data set of single tokens is about 1/4th the size of the single tokens from Direct Supply's DSSI eProcurement system's search queries which has already been benchmarked. I would hypothesize that the below cell takes about **45 ms** to execute.

In [15]:
```
%%time

df['none_values'] = df['letters_only'].apply(lambda s: None if s == '' else s)

df.head()
```

```
CPU times: user 44.6 ms, sys: 7.56 ms, total: 52.1 ms
Wall time: 51 ms
```

Out[15]:

	single_tokens	lower_case	letters_only	none_values
0	tweet	tweet	tweet	tweet
1	!!!	!!!		None
2	RT	rt	rt	rt
3	@mayasolovely:	@mayasolovely:	mayasolovely	mayasolovely
4	As	as	as	as

Removing Empty Values

As mentioned in the previous code cell, once all non-letters where stripped there is the possibility of 'None' values being added to **df**. To remove these *dropna* was used to remove all rows that have an instance of 'None'.

Predictions:

Understanding that the original data set of single tokens is about 1/4th the size of the single tokens from Direct Supply's DSSI eProcurement system's search queries which has already been benchmarked. I would hypothesize that the below cell takes about **160 ms** to execute.

In [16]:
```
%%time

df.dropna(inplace = True)

df.head()
```

```
CPU times: user 131 ms, sys: 12.3 ms, total: 143 ms
Wall time: 144 ms
```

Out[16]:

	single_tokens	lower_case	letters_only	none_values
0	tweet	tweet	tweet	tweet
2	RT	rt	rt	rt
3	@mayasolovely:	@mayasolovely:	mayasolovely	mayasolovely
4	As	as	as	as
5	a	a	a	a

Creating Spell Check Dictionary

The below cell is responsible using the *spell_check_dict* method in order to create a dictionary where the keys are misspelled words and the values are the correctly spelled words.

Predictions:

Understanding that the original data set of single tokens is about 1/4th the size of the single tokens from Direct Supply's DSSI eProcurement system's search queries which has already been benchmarked. I would hypothesize that the below cell takes about **2 s** to execute.

In [17]:
```
%%time

spelling_dict = spell_check_dict(df['none_values'])
```

```
CPU times: user 17.4 s, sys: 55 ms, total: 17.4 s
Wall time: 17.4 s
```

Spell Check Filtering

The below cell creates a new data frame column **spell_checked** from **letters_only** by using the spell check dictionary created in the above cell to ensure words are spelled correctly.

Predictions:

Understanding that the original data set of single tokens is about 1/4th the size of the single tokens from Direct Supply's DSSI eProcurement system's search queries which has already been benchmarked. I would hypothesize that the below cell takes about **82 ms** to execute.

In [18]:
```
%%time

df['spell_checked'] = df['none_values'].apply(lambda s: spell_check(spelling_d
ict, s))

df.head()
```

```
CPU times: user 104 ms, sys: 0 ns, total: 104 ms
Wall time: 103 ms
```

Out[18]:

	single_tokens	lower_case	letters_only	none_values	spell_checked
0	tweet	tweet	tweet	tweet	tweet
2	RT	rt	rt	rt	rt
3	@mayasolovely:	@mayasolovely:	mayasolovely	mayasolovely	mayasolovely
4	As	as	as	as	as
5	a	a	a	a	a

Token Frequency Count: List Method

The below cell uses the method *frequency_dict* in order to create a frequency count of all of the tokens that appear in **spell_checked**, which is then used to call the method *sort_dict* which returns a frequency dictionary where the key is the token from **spell_checked** and the value is the number of times that it appears in that column.

Predictions:

Understanding that the original data set of single tokens is about 1/4th the size of the single tokens from Direct Supply's DSSI eProcurement system's search queries which has already been benchmarked. I would hypothesize that the below cell takes about **70 ms** to execute.

In [19]:
```
%%time

token_list = df['spell_checked'].tolist()

spell_dict = frequency_dict(token_list)

sorted_dict = sort_dict(spell_dict)
```

```
CPU times: user 105 ms, sys: 0 ns, total: 105 ms
Wall time: 106 ms
```

Token Frequency Count: Panda Method

The below cell uses the method *value_counts* which can be called directly on the dataframe column **spell_checked** in order to get the number times that each token appears.

Predictions:

Understanding that the original data set of single tokens is about 1/4th the size of the single tokens from Direct Supply's DSSI eProcurement system's search queries which has already been benchmarked. I would hypothesize that the below cell takes about **35 ms** to execute.

In [32]:
```
%%time

df['spell_checked'].value_counts()
```

```
CPU times: user 49.2 ms, sys: 15 µs, total: 49.3 ms
Wall time: 48.3 ms
```

Out[32]:

a	9498
bitch	8250
rt	7612
i	7530
the	7211
you	6145
to	5349
and	3969
my	3586
that	3586
bitches	3088
in	3070
is	2920
like	2787
me	2738
of	2671
on	2543
be	2378
hoes	2374
all	2177
this	2161
for	2120
pussy	2118
your	2117
im	2090
it	1943
hoe	1912
with	1851
dont	1784
ass	1570
	...
tightly	1
cloprestiwfan	1
senseiquan	1
kolbyxdelce	1
unzip	1
clout	1
honour	1
dunleavy	1
httptcovchxqgmbde	1
gradeyeah	1
nanak	1
avila	1
httptcomjrypfcjpd	1
mrtriskaideka	1
westwestyall	1
rakes	1
worstrapiyrics	1
thatchriseckert	1
saraschaefer	1
kaitlynlardi	1
ajbruno	1
bitchcmon	1
tkncash	1
moneyylol	1
journalists	1
noahcrothman	1

```
shitput                 1
httptcollkuzvtai        1
stripe                  1
seauxcocoa              1
Name: spell_checked, Length: 33426, dtype: int64
```

Token Frequency Count: Functional Programming

The below cell uses the method *frequency_dict* in order to attain a dictionary where the key is the token found in **spell_checked** and the value is the number of times that the token appears in **spell_checked**. Next, a new column **counts** is added to the data frame **df** by applying a lambda that gets the frequency value from the frequency dictionary previously created. Next, duplicates found in the **spell_checked** column are dropped, which creates a new dataframe at the same time, and that new data frame is then sorted in descending order according to **counts**.

In [30]:
```
%%time

df_frequency_dict = frequency_dict(df['spell_checked'])

df['counts'] = df['spell_checked'].apply(lambda s: df_frequency_dict[s])

df_counts = df.drop_duplicates(subset='spell_checked')

df_sorted_counts = df_new.sort_values(by='counts', ascending = False)

df_sorted_counts.head()
```

```
774 ms ± 5.22 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Analysis: Frequency Count Benchmarking

As seen in the above cells, the data frame's built in function of value_counts takes a little over 1/3 of the time required by the equivalent functions needed to convert the data frame into a list, then into a sorted frequency count dictionary. Possible reasons for this time savings is that the latter method requires multiple steps of looping over the entire series of tokens in order to convert it to a list, then again to get token frequency counts, then looping once more over the compressed frequency dictionary in order to sort it. However, the built-in data frame method value_counts is able to create a descending series containing counts of unique values by looping over the complete column of tokens at most one time. By removing a couple of unnecessary steps value_counts is able to only use a single loop, while the equivalent list operation is required to complete three loops. This ratio of 1:3 loops is likely the very reason that the benchmarking times for generating a token frequency count from a data frame is approximately 1/3 of the time required by the equivalent list functions. Finally, there is the functional programming approach to the token frequency count, which took the longest of all of the methods, with a time of 774 ms ± 5.22 ms per loop. The reason for this is that the functional programming approach must first loop over the entirety of the **spell_checked** column to create a frequency dictionary. Then, completely loop back over the **spell_checked** column in order to apply the lambda function that uses the frequency dictionary to return the frequency of the token. Next, a new data frame is created by completely looping over **spell_checked** again in order to drop duplicates. Lastly, another data frame is created by looping over the data frame without duplicates, and sorting it according to the **counts** column in descending order. So, when compared to the built-in panda function *value_counts* and the list method, the functional programming method is comparatively slower because **spell_checked** is looped over completely 3 times, then a smaller sub loop is executed, and all the while two more data frame instances were created which means that every index in the original data frames had to be copied over as well.

Analysis: Reflection on Predictions

Overall, the predictions that I made for the benchmark times of each cell were accurate within a reasonable margin of error. This can be attributed to the fact that the assumption was made that most cells operated with a Big-O notation of O(N), and as such the times could be estimated from the existing body of benchmarked data from Direct Supply's search query data, simply by multiplying by the difference in data set sizes. However, there were two places that this approach proved to be inaccurate. The first was in the method call **spell_check_dict**. In this situation, the hate speech tokens actually took longer to process than Direct Supply's search queries. In hindsight, the likely culprit of this lengthy runtime is the fact that many of the single tokens from the hate speech tweets were lengthy when compared to the search queries. This is due to long twitter handles such as *'@LILBTHEBASEDGOD'*, which when spell checked by the spellchecker library, requires that each character is looped over, and as a result takes much longer to spell check than a four letter word such as 'cake'. The other instance where my benchmark hypothesis was noticeably less accurate than the others was the token frequency count using the List method. My hypothesis was about **30 ms** less than the actual runtime of the cell. It is likely the reason for this is that the runtime of this cell is not O(N), or linear, instead, it is likely closer to being $O(N^2)$, or exponential. This is because there are more than one method call in the cell that requires looping over the complete set of tokens.

Memory Usage: List

The below cell uses the method *getsizeof* to get the memory usage in bytes, of **sorted_dict** the frequency dictionary that was created.

In [21]:
```
import sys

sys.getsizeof(token_list)
```

Out[21]:
```
2726992
```

Memory Usage: DataFrame

The below cell uses the method *memory_usage* to get the memory usage in bytes of each of the columns in the data frame **df**.

In [22]:
```
df.memory_usage(deep=True)
```

Out[22]:
```
Index             2726920
single_tokens    21281775
lower_case       21131863
letters_only     20962636
none_values      20962636
spell_checked    20959156
dtype: int64
```

Reaction: Memory Usage Comparison

The above cells shows a stark difference in the memory usage of a data frame column vs. the list equivalent. It is interesting to see that the data frame column is almost 8 times bigger than the equivalent list. I did not expect such a large difference in the memory usage of the two methods, however, I'm not shocked that it is the data frame column that uses more memory as I would expect there to be extra memory allocated so that each token knows which row its in, as well as which column, and lastly its overall position in the data frame.

Analysis: Overall Panda Usage

Throughout this notebook pandas have been used and also compared to more traditional data structures. Through this work several conclusions can be drawn in terms of the performance of pandas. First of all, it was demonstrated that when creating a frequency count of unique tokens, pandas were significantly quicker than the equivalent functions that used lists when the two were benchmarked against each other for an identical set of tokens. From these findings one can conclude that pandas offer an advantage over built in lists and dictionaries in terms of runtime when the goal is to manipulate a large quantity of similar types of data. Next, it was demonstrated that pandas use much more storage than a equivalent list. In testing, the panda data frame column used about 8 times more storage than the equivalent list. It is likely that this increase in data may be due to information being stored about a given token's row, column, and overall location in the data frame. This extra data may be the very reason that pandas have quicker runtimes for the manipulation of data, with that advantage coming at a cost of memory. So in terms of performance, if you are manipulating data and the use of pandas is being considered, it may be wise to contemplate whether quicker runtimes or decreased memory usage is the priority. However, through the lens of usability pandas offer a clear advantage over built in data structures such as lists or dictionaries. The reason for this is that in a single line of code it is possible to filter, map, and overall manipulate an entire data set, whereas built in data structures require multiple lines of code and/or additional functions in order to execute similar manipulations. However, this shortening of the code comes at a cost of readability. With built in data structures it is often clear what is being done due to the extra length, however, pandas allow for essentially shorthand data manipulation that isn't immediately clear at first glance, especially for the uninitiated.

Conclusion

This notebook is identical to my other notebook **Lab 3: Search Terms with Pandas** in everyway except for the data set that is being used. As such, the same conclusions can be drawn from this notebook, however, by using a different dataset and making runtime predictions, conclusions can now be made about the runtime of common functions to manipulate data using pandas. The data set that these predictions were based on was the data set provided by Tom Davidson's (t-davidson) GitHub repository *hate-speech-and-offensive-language* (https://github.com/t-davidson/hate-speech-and-offensive-language) where he supplies a .csv file of tweets that have been flagged as hate speech. The predictions were made based on the benchmark results from the same functions running on the data set of search queries from Direct Supply's DSSI eProcurement system. Most all predictions were accurate within a reasonable margin of error, with the exception of cells that in hindsight do not have a linear runtime. It is because of this that the conclusion can be made that most of data manipulation done on pandas are executed in a linear runtime with a Big-O notation of O(N) with **N** being the number of elements in the column being manipulated.