Lab 3: Search Terms with Pandas

Introduction:

This notebook's purpose is to build on the work of my previous notebook **Lab 2: Search Terms** where the purpose was to collect a mass amount of search queries, in this case from a .csv file, and filter those search queries into strictly single terms containing no spaces. Once this is completed a frequency dictionary is created from this list of searched tokens and that dictionary is sorted from most searched terms to least searched terms. Next, A second frequency dictionary is created, however, this one is created from the same list that has been spell checked using the *spellchecker* library, and also strictly contains the letters a-z. However, this notebook leverages pandas in order to get the same results as **Lab 2: Search Terms**. Pandas are leveraged against the same problem so that comparisons can be made to built-in data structures in order to understand the strengths and weaknesses of both methods of data manipulation.

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 Direct Supply's DSSI eProcurement system (www.dssi.net) (https://www.dssi.net/). This ecommerce platform is used by Long Term Care and Assisted Living facilites to purchase consumable items. This platform is used by 50,000 distinct users and each one uses DSSI to search for items that they need to buy for their facilities. This specific data set contains 1,048,576 search queries entered by food-service users over a 60 day period from mid 2019.

In [9]:
```
import csv

with open('/data/cs2300/L2/searchTerms.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 in DSSI search queries to be:

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

In [10]:
```
#1
print('Benchmarking for step #1:')
%time filtered_data = filter_rough_data(rough_data, 0)

#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 88.2 ms, sys: 183 µs, total: 88.4 ms
Wall time: 86.8 ms

Benchmarking for step #2:
CPU times: user 154 ms, sys: 532 µs, total: 155 ms
Wall time: 155 ms

Benchmarking for step #3:
CPU times: user 293 ms, sys: 11.9 ms, total: 305 ms
Wall time: 304 ms
```

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 takes the **single_tokens** list and creates a single columned data frame **df**

In [12]:
```
%%time

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

```
CPU times: user 143 ms, sys: 11.9 ms, total: 155 ms
Wall time: 152 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.

In [13]:
```
%%time

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

df.head()
```

```
CPU times: user 259 ms, sys: 31.8 ms, total: 291 ms
Wall time: 289 ms
```

Out[13]:

	single_tokens	lower_case
0	SearchTerm	searchterm
1	36969	36969
2	CMED	cmed
3	500100	500100
4	KEND	kend

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.

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 1.38 s, sys: 11.9 ms, total: 1.39 s
Wall time: 1.39 s
```

Out[14]:

	single_tokens	lower_case	letters_only
0	SearchTerm	searchterm	searchterm
1	36969	36969	
2	CMED	cmed	cmed
3	500100	500100	
4	KEND	kend	kend

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.

In [15]:
```
%%time

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

df.head()
```

```
CPU times: user 186 ms, sys: 4.09 ms, total: 190 ms
Wall time: 189 ms
```

Out[15]:

	single_tokens	lower_case	letters_only	none_values
0	SearchTerm	searchterm	searchterm	searchterm
1	36969	36969		None
2	CMED	cmed	cmed	cmed
3	500100	500100		None
4	KEND	kend	kend	kend

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'.

In [16]:
```
%%time

df.dropna(inplace=True)

df.head()
```

```
CPU times: user 513 ms, sys: 63.8 ms, total: 577 ms
Wall time: 574 ms
```

Out[16]:

	single_tokens	lower_case	letters_only	none_values
0	SearchTerm	searchterm	searchterm	searchterm
2	CMED	cmed	cmed	cmed
4	KEND	kend	kend	kend
6	CMED	cmed	cmed	cmed
8	DYNC1815H	dync1815h	dynch	dynch

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.

In [17]:
```
%%time

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

```
CPU times: user 5.27 s, sys: 28.4 ms, total: 5.29 s
Wall time: 5.29 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.

In [18]:
```
%%time

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

df.head()
```

```
CPU times: user 329 ms, sys: 199 µs, total: 329 ms
Wall time: 327 ms
```

Out[18]:

	single_tokens	lower_case	letters_only	none_values	spell_checked
0	SearchTerm	searchterm	searchterm	searchterm	searchterm
2	CMED	cmed	cmed	cmed	med
4	KEND	kend	kend	kend	end
6	CMED	cmed	cmed	cmed	med
8	DYNC1815H	dync1815h	dynch	dynch	lynch

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.

In [19]:
```
%%time

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

spell_dict = frequency_dict(token_list)

sorted_dict = sort_dict(spell_dict)
```

```
CPU times: user 281 ms, sys: 7.87 ms, total: 289 ms
Wall time: 287 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 of times that each token appears.

In [20]:
```
%%time

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

```
CPU times: user 120 ms, sys: 4.01 ms, total: 124 ms
Wall time: 123 ms
```

Out[20]:
```
chicken       19230
cream         16057
cheese        14014
beef          13566
juice         11488
pie           11475
sauce         11296
pork          11104
potato        10237
green          9804
diced          9610
beans          9139
tomato         8607
corn           8188
apple          7844
bread          7781
sausage        7464
mix            7016
cake           7009
sugar          6505
turkey         6491
onion          6407
rice           6356
bean           6266
bacon          6201
salad          5953
egg            5932
orange         5739
fruit          5626
msc            5602
              ...
sccwp             1
approved          1
obs               1
hoigie            1
crwbkaf           1
glade             1
gerichair         1
gcc               1
fabulous          1
ambassador        1
macao             1
faxed             1
tight             1
residents         1
tbvw              1
tldw              1
lavalosso         1
marainara         1
crayons           1
siligentle        1
cornichon         1
hepmn             1
frenchfries       1
rains             1
mdtbtg            1
tajan             1
```

```
spreader          1
latent            1
mdttbb            1
incident          1
Name: spell_checked, Length: 10605, 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 [24]:
```
%%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_counts.sort_values(by='counts', ascending = False)

df_sorted_counts.head()
```

```
CPU times: user 782 ms, sys: 41.3 ms, total: 823 ms
Wall time: 821 ms
```

Out[24]:

	single_tokens	lower_case	letters_only	none_values	spell_checked	counts
38	chicken	chicken	chicken	chicken	chicken	19230
394	cream	cream	cream	cream	cream	16057
23	cheese	cheese	cheese	cheese	cheese	14014
354	beef	beef	beef	beef	beef	13566
115	JUICE	juice	juice	juice	juice	11488

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.

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 [22]:
```
import sys

sys.getsizeof(token_list)
```

Out[22]: 10436248

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 [23]:
```
df.memory_usage(deep=True)
```

Out[23]:
```
Index            10436176
single_tokens    81906095
lower_case       81890623
letters_only     81620132
none_values      81620132
spell_checked    81608529
counts           10436176
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 data structures 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 notebooks purpose was to build on the work of my previous notebook **Lab 2: Search Terms**. However, this notebook leverages pandas in order to get the same results as the aforementioned search terms lab. Pandas were leveraged against the same problem so that comparisons can be made to built-in data structures in order to understand the strengths and weaknesses of both methods of data manipulation. It was discovered through testing that pandas have favorable runtimes for applying a filter or transformation to large data sets when compared to built-in data structures. However, it was also discovered through testing that pandas use much more data to store the same information as built-in data structures. So in terms of performance, it is important to understand the runtime vs. memory usage tradeoff that occurs when using pandas. In terms of usability, it was discovered that pandas offer a coding advantage after it was observed how multiple lines of data manipulation using built in data structures can be mirrored by a single line of code using pandas. However, this resulting shorthand code is more difficult to understand for an outside observer than the longer equivalent operations accomplished with built-in data structures. To sum up the findings of this notebook, it was discovered that in terms of data manipulation, pandas offer runtime and codability advantages, at the cost of memory usage and readability when compared to built-in data structures.