Lab 2: Search Terms

Introduction:

This NoteBooks purpose is 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.

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
```

Filter: To Lowercase

The cell below contains a method that takes a list of tokens that may contain diferrent cases of letters, and returns a list where every token only contains lower-case letters.

In [4]:
```
def to_lowercase(data):

    lowercase = []
    for x in data:
        lowercase.append(x.lower())

    return lowercase
```

Filter: Remove Non-Letters

The below cell contains a method that recieves a list of search queries that may contain numerical characters and punctuation marks, and returns a list that only contains non-empty tokens comprised of A-Z letters.

In [5]:
```
import re

def remove_non_letters(data):

    only_letters_list = []

    for x in data:
        if not x.isalpha():
            new_token = re.sub(r'[^A-Za-z]','',x)
            if new_token:
                only_letters_list.append(new_token)
        else:
            only_letters_list.append(x)

    return only_letters_list
```

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
```

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 [8]:
```
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 [9]:
```
def spell_check(corrective_spelling_dict, word):

    if word in corrective_spelling_dict:
        return corrective_spelling_dict[word]
    else:
        return word
```

Filter: Spell Check

The cell below contains a method that uses a spell check dictionary (*key = mispelled word, value = correctly spelled word*) and a list of words, and returns a new list where each word is spelled correctly according to the provided spell check dictionary.

In [10]:
```
def spell_check_list(corrective_spelling_dict, words):

    spell_checked = []

    for x in words:
        spell_checked.append(spell_check(corrective_spelling_dict, x))

    return spell_checked
```

Create: .CSV File

The cell below contains a method that takes a file name, and a frequency dictionary (*key = search query, value = # of times that query was searched*) and writes all of the data in the frequency dictionary to the specified file name as a .csv file.

In [11]:
```
def write_csv(file_name, frequency_dict):

    frequency_list = []

    for key, value in frequency_dict.items():
        frequency_list.append([key, value])

    with open(file_name, 'w') as csv_file:
        csv_writer = csv.writer(csv_file)
        csv_writer.writerow(['Token', 'Number of Occurrences'])
        csv_writer.writerows(frequency_list)
```

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 is the search queries entered by food-service users over a 60 dat period from mid 2019.

In [12]:
```
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
4. Filtered to only contain lower-case letters
5. Converted into a dictionary where the key is each search query and the value is the number of times that query appeared in the filtered list.
6. Sorted in decending order from most searched tokens to least searched tokens.
7. Printed for reference.

In [13]:
```
#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)

#4
print('\nBenchmarking for step #4:')
%time lowercase_tokens = to_lowercase(single_tokens)

#5
print('\nBenchmarking for step #5:')
%time frequency = frequency_dict(lowercase_tokens)

#6
print('\nBenchmarking for step #6:')
%time sorted_dict = sort_dict(frequency)
```

```
Benchmarking for step #1:
CPU times: user 70.4 ms, sys: 4.41 ms, total: 74.8 ms
Wall time: 74 ms

Benchmarking for step #2:
CPU times: user 148 ms, sys: 0 ns, total: 148 ms
Wall time: 148 ms

Benchmarking for step #3:
CPU times: user 225 ms, sys: 27.6 ms, total: 253 ms
Wall time: 253 ms

Benchmarking for step #4:
CPU times: user 163 ms, sys: 20.4 ms, total: 183 ms
Wall time: 183 ms

Benchmarking for step #5:
CPU times: user 317 ms, sys: 3.63 ms, total: 321 ms
Wall time: 320 ms

Benchmarking for step #6:
CPU times: user 260 ms, sys: 3.97 ms, total: 264 ms
Wall time: 264 ms
```

Print Data for Reference:

In [14]:
```
# print(sorted_dict)
```

Comments on the Resulting Data:

In general I found the most commonly searched terms rather discouraging for the represented long term care and assisted living facilities. The reason for this it that the majority of the most searched terms are relatively unhealthy. If I had a loved one in one of these facilites, it would be likely that they are not in the best of health, and as such I would not want the majority of their caloric intake to be coming from 'cream', 'cheese', 'pie', 'juice', or one of the many other unhealthy common search queries.

Comments on Nonwords in Data:

With minimal scrolling in the displayed data it becomes clear that there are many nonwords in the data set. It is likely that these nonwords stem from a possible standardized product description system, where these nonwords represent a barcode number, a SKU number, or a UPC number.

Executing Further Filters on Data Set

The cell below picks up on step #4 of the previous code cell and results in the previous data being:

1. Filtered to only contain letter (A-Z).
2. Used to create a spelling dictionary.
3. Filtered according to the created spelling dicitonary.
4. Converted into a dictionary where the key is each search query and the value is the number of times that query appeared in the filtered list.
5. Sorted in decending order from most searched tokens to least searched tokens.
6. Printed for reference.

In [15]:
```
#5
print('Benchmarking for step #5:')
%time letters_only = remove_non_letters(lowercase_tokens)

#6
print('\nBenchmarking for step #6:')
print(len(letters_only))
%time spelling_dict = spell_check_dict(letters_only)

#7
print('\nBenchmarking for step #7:')
%time clean_tokens = spell_check_list(spelling_dict, letters_only)

#8
print('\nBenchmarking for step #8:')
%time clean_frequency = frequency_dict(clean_tokens)

#9
print('\nBenchmarking for step #9:')
%time sorted_clean_tokens = sort_dict(clean_frequency)
```

```
Benchmarking for step #5:
CPU times: user 468 ms, sys: 11.9 ms, total: 480 ms
Wall time: 479 ms

Benchmarking for step #6:
1304522
CPU times: user 5.41 s, sys: 51.9 ms, total: 5.46 s
Wall time: 5.46 s

Benchmarking for step #7:
CPU times: user 183 ms, sys: 24 ms, total: 207 ms
Wall time: 207 ms

Benchmarking for step #8:
CPU times: user 227 ms, sys: 0 ns, total: 227 ms
Wall time: 227 ms

Benchmarking for step #9:
CPU times: user 97.3 ms, sys: 0 ns, total: 97.3 ms
Wall time: 97.1 ms
```

Print Spell Checked Data for Reference:

In [16]:
```
# print(sorted_clean_tokens)
```

Creating a .CSV File from the Data Set

The below cell contains the call to the method that creates a .csv file from the dictionary that remains after step #9 of the previous cell.

In [17]:
```
write_csv('frequency.csv', sorted_clean_tokens)
```

Analysis of Results

By comparing the above spell checked results to the non spell checked results it is immediately clear that some common terms were searched more frequently when all search queries were corrected for spelling as well as punctuation marks. In terms of eliminating punctuation marks and numbers, applying this sort of filter has increased the accuracy of the word related results. The reason for this is that this filter gets rid of arbitrary differences such as plurality and distills the results down to the most common core words that are being searched for. However, eliminating numbers may be reducing the accuracy of the results because it may be much quicker for a customer to search by a SKU number or a similar numerical abbreviation, and now this possibly common search query is not represented in the data. The spell check filtering has similarly conflicting results. For the case where search queries were mispelled by a single letter such as searching for 'chickenx' when the desired search was 'chicken', this filter increases the accuracy of the results because it is a better reflection of desired search queries. However, this elementary spell checking operates by comparing mispelled words to a list of commonly mispelled words, and through experimentation, it is clear that the list of commmonly mispelled words does not often contain brand names. As such, if a user was trying to look for the brand name 'iCookies' the spell check will most likely correct to 'cookies' and thus be a less accurate representation of what the user was attempting to search for.

Analysis of Benchmarking:

After benchmarking all of the method calls, it is clear that the method **spell_check_dict()** is the slowest running method of this notebook. When running on a dual CPU core from the cluster, all other method calls consistently take less than half a second to execute, however, when **spell_check_dict()** uses **distance=1** its execution time is 5.23 s ± 23 ms per loop (mean ± std. dev. of 7 runs, 1 loop each). The reason this method takes the longest to execute is because it uses the Levenshtein Distance (https://en.wikipedia.org/wiki/Levenshtein_distance) algorithm to find permutations within a distance of 1 from the word attempting to be spell checked. It then compares all of these possible permutations to a word frequency list where it selects the most commonly occuring permutation instance from the list. At the root of its comparitively slow execution time is the fact that it must not only loop through the list of words it is attempting to spell check, but for each word that it can spell check, it then must loop through the letters in that word, and finally loop through a sublist to find the most frequently used permutation instance.

Big-O Analysis

After digging into the logic of *spellchecker* which is the brains behind the **spell_check_dict()** method, it is clear why it is the slowest running cell. In a worst case scenerario, each word in the search queries data loaded in this notebook is looped through **n** times, and for each one of those words, *spellchecker* loops through that word's letters **x** times to find permutations of that word, and then *spellchecker* loops through the permutations **z** times to find the most common occurring permutation of the word in its word frequency list. Even though there are multiple loops happening within the largest **n** loop, they are not of the same data set and as such they cannot be multiplied to **n** to increase it to a term that is anything but linear. To sum it up, the Big-O notation of **spell_check_dict()** is:

O(N)

The cell below uses this fact that the Big-O notation is O(N), and the fact that the data set used to benchmark the runtime of **spell_check_dict()** was a size of 1,304,522, in order to calculate a constant to estimate the runtime of different factor sizes of the original data set. Specifically, the below cell makes calls to **time_estimate()** that calculate the runtime for a data set that is 10 times larger, and 100 times larger, respectively.

In [18]:
```
import datetime

def time_estimate(data_size_difference):

    original_data_size = 1304522
    original_runtime = 5.23
    time_factor = original_runtime/(original_data_size)
    return time_factor*(data_size_difference*original_data_size)

print('Estimated runtime of a data set 10x larger than the orginal:')
print(datetime.timedelta(seconds = time_estimate(10)))

print('Estimated runtime of a data set 100x larger than the orginal:')
print(datetime.timedelta(seconds = time_estimate(100)))
```

```
Estimated runtime of a data set 10x larger than the orginal:
0:00:52.300000
Estimated runtime of a data set 100x larger than the orginal:
0:08:43
```

Testing 10x and 100x Data Set

The below cell uses the previous list of cleaned words **letters_only** from the DSSI eProcurement system search queries, and then creates a data set that is ten times and one-hundred times larger by looping over that list 10x or 100x times, and for each loop, either adding the word and a random letter, the word minus its last letter, or the original word. Next, **spell_check_dict()** is called on these created lists in order to test the runtime hypothesis from the above cell.

In [19]:
```
import random
import string

ten_times_larger = []

for x in range(10):
    for word in letters_only:
        i = random.randint(0,2)
        if i == 0:
            ten_times_larger.append(word[:-1])
        elif i == 1:
            ten_times_larger.append(
            word + str(random.choice(string.ascii_letters))
            )
        else:
            ten_times_larger.append(word)

one_hundred_times_larger = []

for x in range(100):
    for word in letters_only:
        i = random.randint(0,2)
        if i == 0:
            one_hundred_times_larger.append(word[:-1])
        elif i == 1:
            one_hundred_times_larger.append(
            word + str(random.choice(string.ascii_letters))
            )
        else:
            one_hundred_times_larger.append(word)

%time ten_x_spelling_dict = spell_check_dict(ten_times_larger)
%time hundred_x_spelling_dict = spell_check_dict(one_hundred_times_larger)
```

```
CPU times: user 1min 42s, sys: 964 ms, total: 1min 43s
Wall time: 1min 43s
CPU times: user 5min 53s, sys: 8.94 s, total: 6min 2s
Wall time: 6min 2s
```

Conclusion

The purpose of this notebook was to compile search query data down into meaningful tokens and know the frequency of those search queries. Through this process it was discovered that food service users of Direct Supply's DSSI eProcurement system frequently searched for items that contained numbers and punctuation marks, and by removing these occurrences it may have made the data more accurate in terms of words that were being searched for, but it negated many data point that could lend insight into the search tool and how its used. In addition, implementing a spell check on a search query data set also increased the accuracy of the spelling for searched for tokens, however, it also negated meaningful data points that users did not mistakenly enter into the search. Lastly, it was hypothesized that implementing *spellchecker* resulted in the method **spell_check_dict()** having a big-O notation of **O(N)**, which when tested, created inconclusive results. It likely that the repetition of adding similar if not identical words to a list does not directly mirror a .csv file that is 10x and 100x larger, and because of this the results from the test appear to be skewed. The prediction for the file that was 10x larger was about 1 minute faster than the test's outcome, however, the prediction for the file that was 100x was about 3 minutes slower than the test's outcome. Due to the fact that the list creation skewed the results, and that the results were much closer than if the big-O was estimated to be O(log(N)) or $O(N^2)$, it is still likely that the big-O of **spell_check_dict()** is **O(N)**.