Mattia Danese
CS 119 - Big Data
Professor Singh
Quiz 3: Hadoop for Fun and Profit

Question 1

1. I defined the Python function below:

```
import numpy as np
import pandas as pd

def make_matrix():
   matrix = np.zeros([77, 77], dtype=float)

   # pickup_community_area, dropoff_community_area, trips
   f = open("chicago_taxi_trips.txt")
   data = f.read()
   f.close()

   lines = data.split('\n')
   for line in lines:
       pickup, dropoff, num_trips = line.split(',')

       if pickup and dropoff and num_trips:
           matrix[int(pickup)-1][int(dropoff)-1] = float(num_trips)
   df = pd.DataFrame(matrix,
                     columns=[i for i in range(77)],
                     index=[i for i in range(77)])
   print(df)

   return matrix
matrix = make_matrix()
```

The output is:

```
~/Desktop/CS 119/quizzes/quiz3 🍣 python3 q1.py
       0       1       2       3       4       5       6       7       8       9       10      11  ...  65  66  67  68  69  70  71  72  73  74      75      76
0   110697   40535   25228    8677    3347   23438    8177   25011      82     397     680     613  ...  23  17  29  36  41  20  17   8   7   5   13054   46960
1    28110  169484   12978   17694    3473    8923    4152   11866      91     666    1034    1331  ...  21   8  37  33  11  24   3  10   0   6   10569   29593
2    41575   22064  378179   63660   38167  314318   84149  185113     148     987    3386    1132  ...  79 141 170 199  58 103  37  40  30  16   39343  118773
3     9768   28753   43074  100298   21596   48395   17387   31547     272     778    2919    1313  ...  18  13  17  28  16  16   5   8   1   9   19050   30255
4     5228    5752   34371   30060   84667  126129   48389   65434     223     889    1152     621  ...  19  16  37  19   7  12  24   4   4  22   27593   13478
..     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...  ...  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..     ...     ...
72       1       2      22       0       2      14      20     278       0       2       0       2  ...  12  20  52  62  19  94  37 938   3  40      52       3
73       0       1       1       0       3       2       3      17       0       0       0       1  ...   0   0   1   4   5   4  19   2 198  19      17       2
74       2       3      13       0       2       9       9      55       0       0       0       0  ...  10  15  22  21   9  43  59  43  25 635      53       6
75   35848   30926   75845   52146   70526  300549  255075 1427825    7702   36006   12237   15308  ...  412 195 246 468 545 306 1171 211 293 458  590584   71246
76   69417   44670  122363   48455   14384  114834   33617  104935     135     773    3098    2036  ...  26  76  62  82  26  80  23  62   6  10   35950  277895

[77 rows x 77 columns]
~/Desktop/CS 119/quizzes/quiz3 🍣
```

2. Results for 2 iterations:

Rank	Area	TrafficRank
1.0	7.0	0.3926022090534061
2.0	31.0	0.2584648119826917
3.0	27.0	0.09962175754875345
4.0	75.0	0.05002224141765879
5.0	6.0	0.04917004921567052
6.0	5.0	0.0472410765971285
7.0	23.0	0.02821808696003908
8.0	32.0	0.024876284609468495
9.0	55.0	0.014747024385052638
10.0	2.0	0.00843482373777415

Results for 4 iterations:

Rank	Area	TrafficRank
1.0	7.0	0.39691856933959946
2.0	31.0	0.26261388644282685
3.0	27.0	0.10110830641750371
4.0	75.0	0.050134346418633624
5.0	6.0	0.04700794222225864
6.0	5.0	0.042716766984098216
7.0	23.0	0.027282934130077516
8.0	32.0	0.025414250139712043
9.0	55.0	0.014816616641869259
10.0	2.0	0.007470466893208742

Results for 8 iterations:

Rank	Area	TrafficRank
1.0	7.0	0.3969714161723048
2.0	31.0	0.26268659963543767
3.0	27.0	0.10114122838482863
4.0	75.0	0.05013562929519193
5.0	6.0	0.046963217417118175
6.0	5.0	0.04263967280637705
7.0	23.0	0.027270030158624652
8.0	32.0	0.025426027377613448
9.0	55.0	0.014817445598216515
10.0	2.0	0.007456756836265385

Results for 64 iterations:

Rank	Area	TrafficRank
1.0	7.0	0.3969714308094607
2.0	31.0	0.2626866218079573
3.0	27.0	0.10114123891356738
4.0	75.0	0.05013562944357687
5.0	6.0	0.046963203556543126
6.0	5.0	0.042639650007880614
7.0	23.0	0.02727002625622229
8.0	32.0	0.02542603104553855
9.0	55.0	0.014817445805868334
10.0	2.0	0.007456752885205922

The plots above show the ten areas that have the highest TrafficRank after 2, 4, 8, and 64 iterations.
The plots above clearly show that the TrafficRank values converge after 4 iterations.

I defined the Python function below:

```
def run_TrafficRank(matrix):
    iterations = [2, 4, 8, 64]

    for iteration in iterations:
        m = np.copy(matrix)
        ranks = np.ones((77,1))/77.0
        n_vector = np.ones((77,1))/77.0

        # run the algorithm for 'iteration' times
        for i in range(iteration):
            ranks = np.matmul(0.85 * m, ranks) + 0.15 * n_vector
            ranks = ranks / sum(ranks)

        # sorting areas based on ranks
        sorted_ranks = []
        for i in range(77):
            sorted_ranks.append((i, ranks[i][0]))
```

```
            sorted_ranks = sorted(sorted_ranks, key=lambda x:-x[1])

            # getting top 10 areas based on rank
            output = []
            for idx, i in enumerate(sorted_ranks[:10]):
                output.append((idx+1, i[0], i[1]))

            # plotting table
            df = pd.DataFrame(output,
                              columns=["Rank", "Area", "TrafficRank"])
            fig, ax = plt.subplots(1,1)
            table = ax.table(cellText=df.values,
                             colLabels=df.keys(),
                             loc='center')
            ax.axis('off')
            ax.axis('tight')
            ax.set_title("{} Iterations".format(iteration))
            for key, cell in table.get_celld().items():
                if key[1] == 0 or key[1] == 1:
                    table.get_celld()[key].set_width(0.1)
            plt.show()

    matrix = make_matrix()
    run_TrafficRank(matrix)
```

3. My calculations of TrafficRank correspond fairly well with the Hardship Index. Areas 32, 6, 7, 5, and 2 are all in the top 10 for the Hardship Index and they are all also in the top 10 for my TrafficRank calculations. Additionally, the relative order of these five areas is also somewhat similar: area 2 was the lowest and area 5 has the 5th highest rank in both the Hardship Index and my TrafficRank calculations. That being said, there is definitely some variance between the Hardship Index and my TrafficRank calculations: the highest area in the Hardship Index (area 8) does not show up in my top 10 for TrafficRank and area 32 was the 2nd highest for the Hardship Index but ranks 8th highest for TrafficRank.

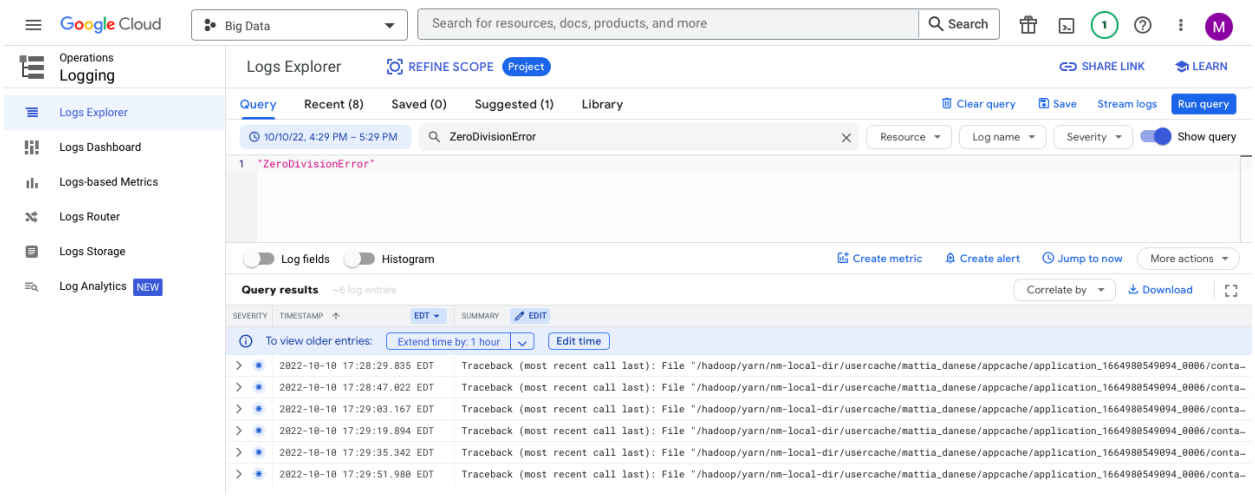
Question 2

1. 6 divide-by-zero errors occurred and they happened on the `us-central1-c` server location.

2. I found 6 such divide-by-zero error messages.
   No, the count is not consistent with what I expected from `random.randint(0,99)`. Since 0 has a 1% chance of getting chosen from `random.randint(0,99)`, then each time a word is read there is a 1% of a divide-by-zero error occurring. Thus, I was expecting the amount of divide-by-zero errors to be roughly 1% of all words; however, 6 is drastically less than that.

The commands I ran:

- `git clone https://github.com/singhj/big-data-repo.git`
- `cd big-data-repo/`
- `hadoop fs -mkdir /five-books`
- `hadoop fs -put ./* /five-books`
- `hadoop fs -put hadoop_error_mapper.py /`
- `hadoop fs -chmod a+x /hadoop_error_mapper.py`
- `hadoop jar /usr/lib/hadoop/hadoop-streaming.jar -files hadoop_error_mapper.py -mapper hadoop_error_mapper.py -reducer aggregate -numReduceTasks 1 -input /five-books/five-books -output /books-count-error-mapper`
- I then went back to the Google Cloud Console, clicked on “View Logs”, and searched for “ZeroDivisionError” in the search field of the “Logs Explorer” tab



Question 3

I defined the Python function below:

```
def fcat(*args):
   if len(args) == 1:
       return str(args[0])
   return str(args[0]) + "-" + fcat(*args[1:])

print(fcat("red", "fish", "blue", "fish"))
print(fcat(*[i for i in range(100)]))
print(fcat(["hello"], ['world'], "!", 123, ['a', 2, ['c']]))
```

The output is:

```
red-fish-blue-fish
0-1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24-25-2
6-27-28-29-30-31-32-33-34-35-36-37-38-39-40-41-42-43-44-45-46-47-48-4
9-50-51-52-53-54-55-56-57-58-59-60-61-62-63-64-65-66-67-68-69-70-71-7
2-73-74-75-76-77-78-79-80-81-82-83-84-85-86-87-88-89-90-91-92-93-94-9
5-96-97-98-99
['hello']-['world']-!-123-['a', 2, ['c']]
```

Question 4

1. The two Presidents that have the highest valence per 1,000 words are: Eisenhower and Taylor

2. The President that has the lowest valence per 1,000 words is: Johnson

3. Assuming there are 43 Presidents and each President has on average $19.7 \approx 20$ speeches.
   (43 presidents) * (20 speeches) * (20 bytes per president name + 8 bytes per integer) = 24,080 bytes
   Thus, 24 KB of data would flow through the shuffle network.

I composed the following script. The script first loads in all the words that have a valence and stores them in a dictionary. In my mapper section, I loop through every speech of every president. For each speech, I sum up its valence, divide its valence by the total number of words, multiply this quotient by 1000 (to get valence per 1000 words), and then emit a set containing the president's name and the valecne per 1000 words of the current speech. In my reducer section, I create a dictionary to hold the final valences of each president across all of their respective speeches. I loop through the collection of all emitted sets by the mapper, use the name of the president in the set as the key for the "final valence" dictionary and add the valence per 1000 words in the set to the current president's running total. Finally, I divide the final valence of each president by their respective number of speeches to get the average valence per 1000 words across all of that president's speeches.

```
import os
import string

# configure valence_map
valence_map = {}

f = open("valence_map.txt")
data = f.read().split('\n')
f.close()

for line in data:
   word, valence = line.split('\t')
   word = word.lower() # makes all lowercase
   if word not in valence_map:
       valence_map[word] = int(valence)

# Mapper
mapper_list = [] # will hold (pres_name, valence_per_speech) tuples
```

```
num_words = {} # holds count of number of words across all speeches for
each president
path = "./prez_speeches"

root, dirs, files = next(os.walk(path), (None, None, []))
for d in dirs: # loops through all 'prez_speeches' subfolders
   path = "./prez_speeches/" + d
   root, dirs, files = next(os.walk(path), (None, None, []))
   for file in files: # loops through each speech file for every
president
       f = open(path + "/" + file)
       data = f.read().split('\n')[2:]
       f.close()
       speech_valence = 0
       num_words = 0

       for line in data: # loops over every line of each speech
           if line: # ensure not empty line
               for word in line.split(): # loops over every word in
each line of each speech
                   word = word.translate(str.maketrans('', '',
string.punctuation)) # take out punctuation
                   word = word.lower() # makes all lowercase
                   if word in valence_map: # gets valence of word
                       speech_valence += valence_map[word]
                   num_words += 1

       mapper_list.append((d, (speech_valence / num_words) * 1000.0))

# Reducer
total_valence = {}
num_speeches = {}
for prez, valence in mapper_list:
   # sums up valence of every speech per president
   if prez in total_valence:
       total_valence[prez] += valence
   else:
       total_valence[prez] = valence

   # keeps track of the number of speeches per president
   if prez in num_speeches:
       num_speeches[prez] += 1
   else:
```

```
            num_speeches[prez] = 1

    # get average of all 'valences per 1000 words' for every speech of
    every president
    for prez in total_valence:
       total_valence[prez] /= num_speeches[prez]

    # Printing Results
    total_valence_list = []
    for prez in total_valence.keys():
       total_valence_list.append((prez, total_valence[prez]))
    total_valence_list = sorted(total_valence_list, key=lambda x: -x[1])

    print("The 2 Presidents with the highest valence per 1,000 words are {}
    ({}) and {} ({})".format(total_valence_list[0][0],
    total_valence_list[0][1], total_valence_list[1][0],
    total_valence_list[1][1]))
    print("The President with the lowest valence per 1,000 words is {}
    ({})".format(total_valence_list[-1][0], total_valence_list[-1][1]))

    for i in total_valence_list:
       print("{}: {}".format(i[0], i[1]))
```

The output is:

```
The 2 Presidents with the highest valence per 1,000 words are
eisenhower (65.88670031299279) and taylor (64.63873329468443)
The President with the lowest valence per 1,000 words is johnson
(12.33675832245528)
eisenhower: 65.88670031299279
taylor: 64.63873329468443
truman: 60.381811315215465
coolidge: 57.97535360576024
monroe: 57.93391651427335
ford: 56.73778559661736
carter: 51.1516750486409
washington: 50.791395840016776
obama: 50.163770973152104
nixon: 47.86560826878803
madison: 45.28415269186997
bharrison: 44.91255430924635
adams: 44.643064742477755
jqadams: 44.44367214854559
kennedy: 44.40200363011708
```

```
clinton: 44.23920402355129
bush: 43.69247914124243
jefferson: 43.63816322072355
mckinley: 43.63726217564954
harding: 41.754856733867584
fillmore: 41.26646197939838
lbjohnson: 40.80318895746353
gwbush: 40.708613378219624
garfield: 39.59390862944162
reagan: 38.947042728605844
taft: 38.696620017182106
harrison: 38.074967482558826
grant: 37.601076420401235
wilson: 37.54427214197751
jackson: 36.698531003134036
polk: 34.50759428219121
hayes: 32.205974946900405
arthur: 31.04932039043986
cleveland: 30.494378096239792
pierce: 29.588885549786045
hoover: 26.50038143517079
fdroosevelt: 26.34918394995711
roosevelt: 26.266629863200126
tyler: 24.726406035675424
vanburen: 22.954016110053843
lincoln: 20.074154894406867
buchanan: 13.519353130009298
johnson: 12.33675832245528
```

Note: I am aware that my code runs in local and does not run in Hadoop. The requirement of using Hadoop for this question was only made aware to me, and many others in the class, a mere day before this assignment was due. I am grateful for the one-day extension, however, with such short notice and other assignments due Thursday and Friday, I did not have time to allocate to fixing my code.