**TRAFFICRANK**:

1.

```
import pandas as pd
from scipy import sparse
import numpy as np
# not print scientific notations
pd.options.display.float_format = '{:f}'.format
np.set_printoptions(suppress=True)

df =
pd.read_csv('https://storage.googleapis.com/jsingh-bigdata-public/chicago-taxi-rides.csv')
df.dropna(inplace = True)
df.reset_index(inplace=True, drop=True)
df = df[df.columns].astype(int)
matrix = sparse.coo_matrix((df.trips.values, (df.pickup_community_area-1,
df.dropoff_community_area-1)), (77, 77))
matrix = matrix.toarray()
matrix = matrix.astype('float')

# normalize it by the total distance summed within columns
col_sums = np.sum(matrix, axis = 0)
for i in range(77):
   matrix[:,i] = matrix[:,i]/float(col_sums[i])
```

2. Implement the TrafficRank algorithm

```
init = np.ones((77, 77))/77.0
beta = 0.85

ranks_old = np.ones((77, 1))/77.0
power = [2**i for i in range(1, 7)]
result_rank = list()

# matrix A
A = (beta * matrix) + ((1 - beta)*init)

for i in range(64):
   ranks_old = np.matmul(A,ranks_old)
   if i+1 in power:
       result_rank.append(ranks_old.reshape(77,))
```

```
reshape_result = np.array(result_rank).T
column_names = ["rank " + str(i) for i in power]
result_df = pd.DataFrame(reshape_result, columns = column_names)
result_df
```

✓ 0.8s

| | rank 2 | rank 4 | rank 8 | rank 16 | rank 32 | rank 64 |
|---|---|---|---|---|---|---|
| 0 | 0.006221 | 0.005239 | 0.005016 | 0.005009 | 0.005009 | 0.005009 |
| 1 | 0.007271 | 0.005592 | 0.005217 | 0.005204 | 0.005204 | 0.005204 |
| 2 | 0.016433 | 0.015223 | 0.014841 | 0.014829 | 0.014829 | 0.014829 |
| 3 | 0.007397 | 0.006154 | 0.005911 | 0.005905 | 0.005905 | 0.005905 |
| 4 | 0.007654 | 0.007047 | 0.006930 | 0.006928 | 0.006928 | 0.006928 |
| ... | ... | ... | ... | ... | ... | ... |
| 72 | 0.002987 | 0.002549 | 0.002493 | 0.002492 | 0.002492 | 0.002492 |
| 73 | 0.002104 | 0.002065 | 0.002064 | 0.002064 | 0.002064 | 0.002064 |
| 74 | 0.002511 | 0.002291 | 0.002274 | 0.002273 | 0.002273 | 0.002273 |
| 75 | 0.047121 | 0.046942 | 0.047008 | 0.047017 | 0.047017 | 0.047017 |
| 76 | 0.012620 | 0.011086 | 0.010661 | 0.010647 | 0.010647 | 0.010647 |

77 rows × 6 columns

3. For this question, I print the 10 largest values and 10 smallest values in the rank 64 columns (have to add 1 because the table in the report is 1-indexing). Larger values correspond to lower hardship index, which indicates a healthier economy of the region. I looked at the hardship index and the city names corresponding to my output. The result didn't totally match up with what the report said. First of all, these don't correspond to the top 10 best and worst cities laid out in the chart in the report. However, most of the cities in my top-10 and bottom-10 outputs are correctly in the lower/upper half of the chart, respectively (Near North Side and Loop are correctly identified as the best cities—with lowest hardship indexes). That being said, some ended up in the wrong bucket (having a high hardship index score but is ranked highly in my model). I think the model has not totally converged after 64 iterations.

```
# Indices of N largest elements in list
# using sorted() + lambda + list slicing
rank_64 = result_df['rank 64'].argsort()
print(rank_64[:10] + 1)
print(rank_64[-10:]+ 1)
```

7] ✓ 0.1s

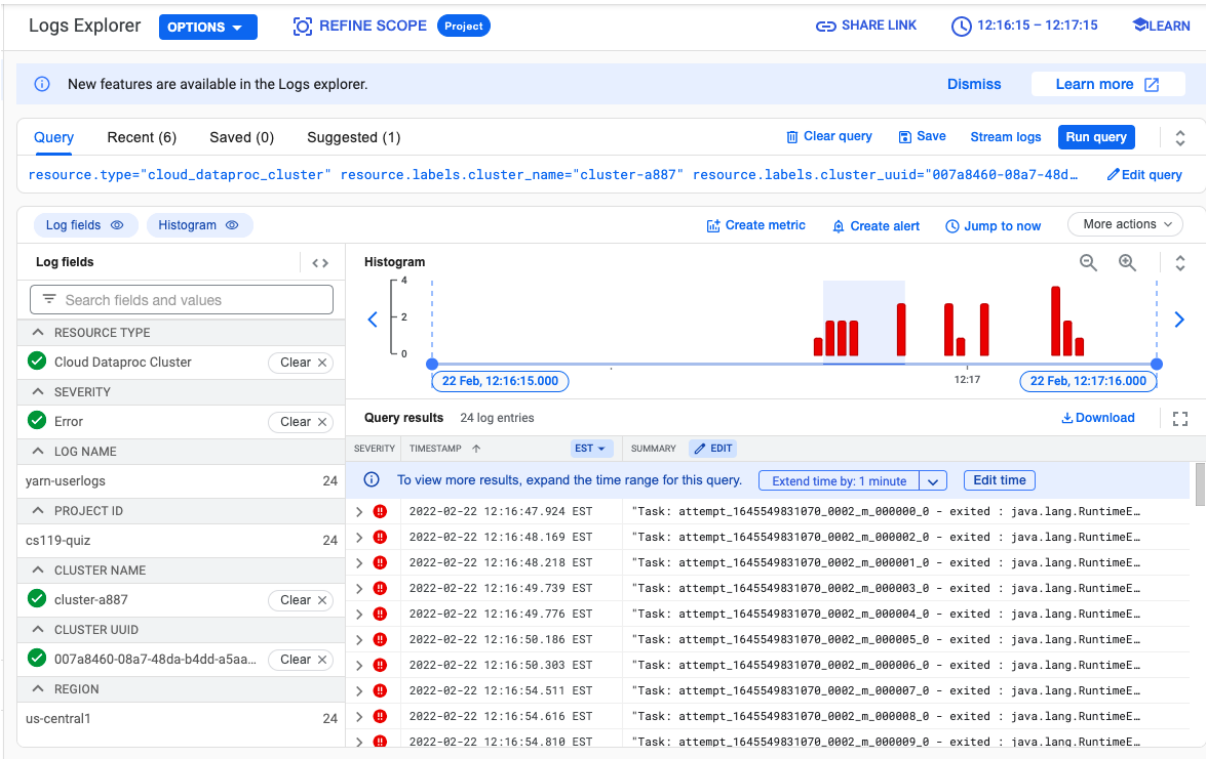
```
0     9
1    74
2    72
3    52
4    18
5    20
6    26
7    75
8    29
9    47
Name: rank 64, dtype: int64
67     3
68    56
69    33
70    24
71     7
72    76
73     6
74    28
75    32
76     8
Name: rank 64, dtype: int64
```

**HADOOP ERRORS**:

For this question, I used the command lines provided in the slides to create the cluster (used 1 worker node in the configuration). I also used vim to copy over the mapper.py script from the doc, and run:

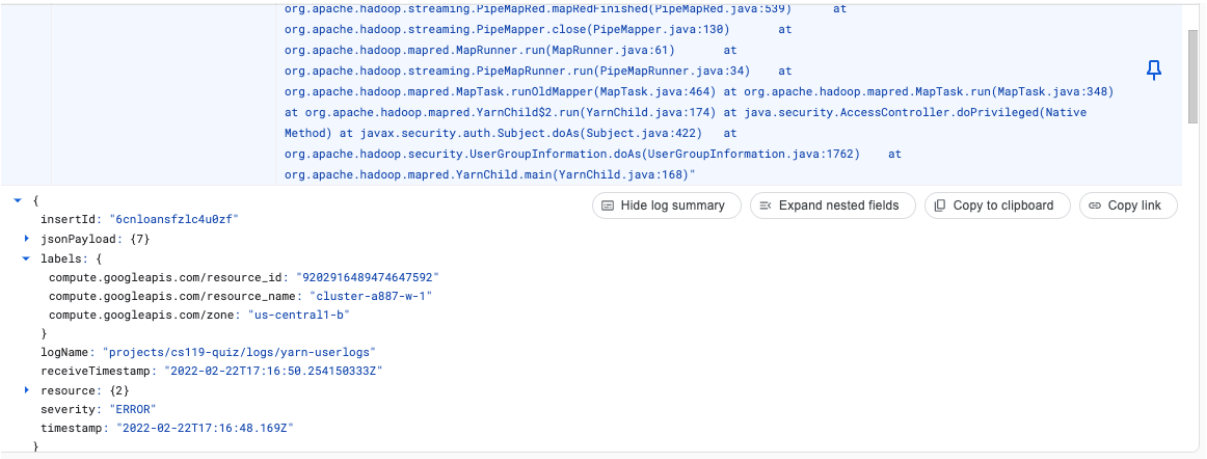
```
hadoop jar /usr/lib/hadoop/hadoop-streaming.jar \
-files mapper.py -mapper mapper.py \
-reducer aggregate \
-input /user/five-books \
-output /user/j_singh/books-count-pymapper
```

Then I went to the “View logs" on the cluster to see the “Errors” (all of them are Java's RuntimeError, which is ZeroDivision in this case), this is what I got:



***Where (what server & location) did the divide-by-zero error messages show up?***

Clicking into one of the errors to see the details, we see that the errors showed up at `us-central1-b`




***How many such messages did you find? Is the count you found consistent with what you might expect from `random.randint(0,99)`?***

There are **24** RuntimeError in total, which is consistent with what you might expect from **random.randint(0,99)**

**PRESIDENTIAL SPEECHES**

Assume each character is 1 byte. Let p be the number of presidents, s_i be the number of speeches for president i, , w_i be the number of words for president i. Also assume that the number of words is much larger than the number of speeches, and the number of speeches is much larger than the number of persons.

**Method 1**: The intermediate value is {(president, speech_name), score} so the rough formula of the data flow is:

20*p*sum(s_i) + speech name bytes + 8*score

Assume that speech name bytes are not substantial since we can always rename the speech down to trivial (single/double characters) names. Thus, this formula grows as the number of presidents and speeches grow (and the number of speeches dominates). Thus, O(number of speeches)

**Method 2**: The intermediate value is {(president, word), score} so the rough formula is:

20*p*sum(w_i) + 8*word + 8*score

With similar logic to method 1, we can see that this is going to be O(number of words), which dominates the number of speeches by our assumption. Also we cannot shrink the words (second term) like we do in speech names, so this definitely will take up a lot more space.

We should choose the first way, emitting the set of (president, aggregate valence) for each president in the dataset. The mapper of the first method generates the set {(president, speech), score} for each president-speech pair, as opposed to {(president, word), score} in method 2, which results in a way larger list, taking up more memory space. The reducer of method 1 now receives a much smaller intermediate output and can process it in less time. Since only the total valence of each president matters, and not the valence of each word/each speech, we should go with the first method.

Mapper:

```
url =
"https://raw.githubusercontent.com/fnielsen/afinn/master/afinn/data/AFINN-en-165.tx
t"
file = urllib.request.urlopen(url)
afinn = dict()
for line in file:
	decoded_line = line.decode("utf-8")
	split = decoded_line.split('\t')
	afinn[split[0]] = int(split[1].strip())

result = dict()
PATH = '/Users/irenechang/Downloads/prez_speeches'
filelist = os.listdir(PATH)
filelist.remove('.DS_Store')
pattern = re.compile("[a-zA-Z][a-zA-Z0-9]*")
for filename in filelist:
	president = filename
	subfilelist = os.listdir(PATH+'/'+filename)
	score = 0
	for subfile in subfilelist: # for every speech.txt
		speech = subfile.split('.')[0]
		f = open(PATH+'/'+filename+'/'+subfile, 'r')
```

```
            while True:
                  line = f.readline()
                  if not line:
                        break
                  for word in pattern.findall(line):
                        if word.lower() in afinn:
                              score = afinn[word.lower()]
                        else:
                              score = 0

                        if (president, speech) in result:
                              result[(president, speech)] += score
                        else:
                              result[(president, speech)] = score
return result
```

Reducer:

```
final = {}
for key in pres_score:
      if key[0] in final:
            final[key[0]] += pres_score[key]
      else:
            final[key[0]] = pres_score[key]

# score per 1000 words:
for president in final:
      final[president] /= 1000

return final
```

***Two Presidents whose speeches had the highest valence per 1,000 words.***

1/ Obama

2/ Lyndon B. Johnson

***The President whose speeches had the lowest valence per 1,000 words.***

Lincoln

Output after mapper and reducer:

```
{'lincoln': -0.146, 'garfield': 0.114, 'harrison': 0.317, 'taylor': 0.588,
'adams': 0.667, 'harding': 0.921, 'buchanan': 1.196, 'eisenhower': 1.2,
'fillmore': 1.398, 'madison': 1.427, 'jefferson': 1.444, 'johnson': 1.499,
'washington': 1.653, 'jqadams': 1.708, 'arthur': 1.804, 'truman': 1.842,
'pierce': 1.942, 'vanburen': 2.067, 'tyler': 2.082, 'ford': 2.26, 'hoover':
2.369, 'nixon': 2.392, 'polk': 2.589, 'bharrison': 2.73, 'monroe': 2.805,
'wilson': 2.86, 'hayes': 3.017, 'fdroosevelt': 3.083, 'grant': 3.602,
'carter': 3.651, 'coolidge': 3.918, 'mckinley': 4.025, 'bush': 4.436,
'gwbush': 4.636, 'taft': 4.916, 'roosevelt': 5.006, 'cleveland': 5.049,
```

'jackson': 5.064, 'kennedy': 6.302, 'clinton': 6.579, 'reagan': 8.278,
'lbjohnson': 10.05, 'obama': 10.385}