Mohammadreza Taremi – 96101902 - Assignment 4

---

1)

a. If there are no rows among k rows which contain a 1 in a specific column, then the aftermath of *minhashing* is ‘don’t know’

Proof:

We have m 1s, therefore n-m zeros.

$$P(don't\ know) = \Pr(number\ of\ chose\ 1s) = \frac{\binom{n-m}{k}}{\binom{n}{k}} = \frac{\frac{(n-m)!}{(n-m-k)!\,k!}}{\frac{n!}{(n-k)!\,k!}}$$

$$= \frac{(n-k)!}{n!}\frac{(n-m)!}{(n-k-m)!}$$

$$= \frac{(n-k)}{n}\frac{(n-k-1)}{n-1}\dots\frac{(n-k-m+1)}{(n-m+1)}$$

We know that each term on $LSH \leq \frac{n-k}{n}$

Therefore we have:

$$\frac{(n-k)}{n}\frac{(n-k-1)}{n-1}\dots\frac{(n-k-m+1)}{(n-m+1)} \leq \left(\frac{n-k}{n}\right)^m$$

b. This algorithm fails if $k < k_0$

k_0 computation:

$$P(don't\ know) \leq \left(\frac{n-k}{n}\right)^m = \left(1-\frac{k}{n}\right)^{\frac{n}{k}\cdot\frac{km}{n}} \sim e^{-\frac{km}{n}} \leq e^{-10} \rightarrow k \geq 10\,\frac{n}{m}$$

2)

a.

$\Pr(having\ minutia\ in\ each\ square|finger\ 1) = (0.2)^3 = 0.008$

$\rightarrow \Pr(finger2) = (1 - 0.2)^3 = (0.8)^3 = 0.512$

$\rightarrow \Pr(both\ are\ in\ the\ bucket|same\ finger) = (0.008)(0.512)$

$= 0.004096$

$\Pr(same\ bucket|same\ fingers) = 1 - (1 - 0.004096)^{2048} = 0.9997$

$\Pr(same\ bucket\ |\ different\ fingers) = 1 - (1 - 0.2^6)^{2048} = 0.122$

$False\ Negative \sim 0.02\%$ $\quad$ $Flase\ Positive \sim 12.2\ \%$

By using AND construction, we can greatly reduce the probability of a false positive, while making only a small increase in the false negative.

$\Pr(matching\ fingerprint) = (0.99)^2 \sim 0.9995 \rightarrow False\ Neg \sim 0.05\%$

$\Pr(False\ Pos) = (0.122)^2 = 0.014 \rightarrow False\ Pos = 1.4\%$

b.

$Flase\ Pos + Flase\ Neg = \ 1 - (1 - 0.004096)^n + \ 1 - (1 - 0.000064)^n$

$= 2 - [(1 - 0.004096)^n + \ (1 - 0.000064)^n\ ]$

This function is strictly ascending, then its minimum is located at in the minimum value of n $\rightarrow 2^1$

3)

a. In this part, I used the codes which belong to the previous assignment.

```
#part a

import pandas as pd
#first of all, we have to load the data
data  = sc.textFile('/content/drive/My Drive/Sampe_Data/Sample_Traffic.csv')
header = data.first()
data = data.filter(lambda line: line != header)
data = data.sample(False,0.1,101)
#this function is capable of splitting columns of data
#['DEVICE_CODE,SYSTEM_ID,ORIGINE_CAR_KEY,FINAL_CAR_KEY,CHECK_STATUS_KEY,COMPA
def parseLine(line):
  fields = line.split(',')
  DEVICE_CODE = int(fields[0])
  ORIGINE_CAR_KEY = int(fields[2])
  PASS_DAY_TIME = pd.to_datetime(fields[6]).day
  return (ORIGINE_CAR_KEY,PASS_DAY_TIME,DEVICE_CODE)

traffic_rdd = data.map(parseLine)
traffic_rdd.first()
#print (200501,1,10477885) tuple

traffic_rdd = traffic_rdd.map(lambda x:(tuple([x[0],x[1]]),x[2]))
traffic_rdd_gb = sorted(traffic_rdd.groupByKey().mapValues(list).collect())
```

```
traffic_rdd_gb
```

Sample output:

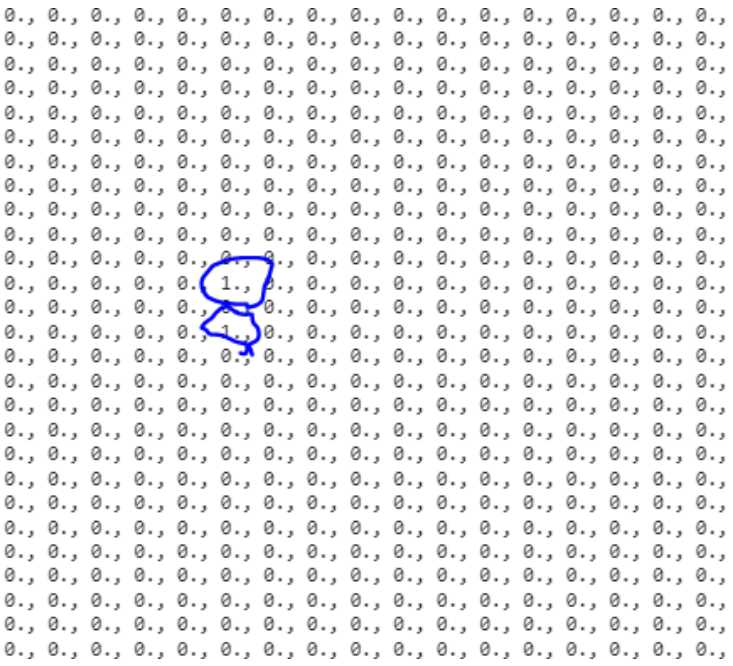
```
((7632039, 1), [208602, 631795, 900246, 100701130])
```

It means that Car with a **7632039** Code has been observed in streets with the codes: **208602, 631795, 900246, and 100701130.**

b. In this part, we have to create a list with the length of all the existed paths. Then, for each record, we create a sparse list of 1s and 0s. If there is a common path, we set the value to 1; otherwise, we set it to 0. We do this by using the following code snippet:

```
import numpy as np
all_paths = list(np.unique(all_paths))
adj_matrix = np.zeros((len(traffic_rdd_gb),len(all_paths)))
for i in range(len(traffic_rdd_gb)):
  for element in list(set(all_paths).intersection(traffic_rdd_gb[i][1])):
    ind = all_paths.index(element)
    adj_matrix[i][ind] = 1
```

Sample output:



It means that in this sample output, we have 2 common paths which also exist in the *all_paths* list.

Now we have to create a list composed of 1s and 0s that are arranged randomly. To do so, we use the following codes:

```
import random
random.seed(101)
random_device_code_list = [0] * random.randint(0,len(all_paths)) + [1]*(len(all_paths) - random.randint(0,len(all_paths)))
random.shuffle(random_device_code_list)
```

Now it's time to compute the Cosine Similarity between the random device code list and each of the rows of *adj_matrix*. We do this by the following codes:

```
from numpy import dot
from numpy.linalg import norm
cos_sim = np.zeros(len(traffic_rdd_gb))
for i in range(len(traffic_rdd_gb)):
  cos_sim[i] = dot(random_device_code_list, list(adj_matrix[i]))/(norm(random_device_code_list)*norm(list(adj_matrix[i])))
cos_sim
```

```
array([0.       , 0.0531494, 0.0531494, ..., 0.       , 0.       ,
       0.       ])
```

And finally, we just have to extract the indices of top 5 elements in *cos_sim* list before printing them:

```
top_5_idx = list(np.argsort(cos_sim)[-5:])
for i in top_5_idx:
  print(traffic_rdd_gb[i])
```

```
((22826618, 1), [206701, 142, 900215, 900234, 900215, 900233, 100700862, 900234])
((59796348, 1), [900202, 100700820, 100700820, 100701096, 900107, 900244, 900208])
((9965926, 1), [900174, 900183, 128, 100700862, 100700820])
((8087050, 1), [900244, 900264, 100700839, 100701092, 900156])
((8970694, 1), [631830, 230204, 100700826, 231, 900156])
```

c. In this part, we are to implement LSH algorithm in terms of **cosine similarity** metric. To do so, we can produce a family of LSH functions compatible with the cosine similarity by considering a family of random hyperplanes. Then, we say two vectors are similar if they lie on a same side of the hyperplane, meaning the dot product of the normal vector of the hyperplane with the vectors are of the same sign:

$$\forall\, V_1, V_2 : V_1 \; and \; V_2 \; are \; similar \mid V_1.n \geq 0 \; and \; V_2.n \geq 0$$

First of all, we compute the cosine similarity between two vectors by using its definition. We also have to create a signature matrix and break it into bands. We do this by the following commands:

```
# Compute signature matrix
R = A@np.random.randn(D, n)
S = np.where(R>0, 1, 0)

# Break into bands
S = np.split(S, b, axis=1)
```

After converting band matrices to band columns we find all the pairs for every bucket. As a matter of fact, these pairs are our LSH candidate pairs. Finally, we perform the actually similarity computation by using *cos_sim* function which we have defined before to find the pairs having a greater cosine similarity than the threshold. The code snippet below does the trick for us:

```
# column vector to convert binary vector to integer e.g. (1,0,1)->
binary_column = 2**np.arange(r).reshape(-1,  1)

# convert each band into a single integer,
# convert band matrices to band columns
S = np.hstack([M@binary_column for M in S])

# Every value in the matrix represents a hash bucket assignment
# For every bucket in row i, add index i to that bucket
d = collections.defaultdict(set)
with np.nditer(S,flags=['multi_index']) as it:
    for x in it:
        d[int(x)].add(it.multi_index[0])

# For every bucket, find all pairs. These are the LSH pairs.
candidate_pairs = set()
for k,v in d.items():
    if len(v) > 1:
        for pair in itertools.combinations(v, 2):
            candidate_pairs.add(tuple(sorted(pair)))
```

Please note that *random_device_code_list* is our abstract path mentioned in the instruction file.

After running this code, Google RAM has depleted and the whole session was crashed. To overcome this problem, I sampled the dataset with the rate of 0.05 instead of 0.001

For this sample rate, the algorithm couldn't find any similar path to the random path.

4)

This algorithm takes ideas from both hierarchical and point-assignment approaches. Like CURE, it represents clusters by sample points in main memory. However, it also tries to organize the clusters hierarchically, in a tree, so a new point can be assigned to the appropriate cluster by passing it down the tree. Leaves of the tree hold summaries of some clusters, and interior nodes hold subsets of the information describing the clusters reachable through that node. An attempt is made to group clusters by their distance from one another, so the clusters at a leaf are close, and the clusters reachable from one interior node are relatively close as well.

In this algorithm we represent the clusters as follows:

As we assign points to clusters, the clusters can grow large. Most of the points in a cluster are stored on disk, and are not used in guiding the assignment of points, although they can be retrieved. The representation of a cluster in main memory consists of several features.

Initializing the Cluster Tree

The clusters are organized into a tree, and the nodes of the tree may be very large, perhaps disk blocks or pages, as would be the case for a B-tree or R-tree,which the cluster-representing tree resembles. Each leaf of the tree holds asmany cluster representations as can fit. Note that a cluster representation hasa size that does not depend on the number of points in the cluster.

Adding Points in the GRGPF Algorithm

We now read points from secondary storage and insert each one into the nearest cluster. We start at the root, and look at the samples of clustroids for each of the children of the root. Whichever child has the clustroid closest to the new point p is the node we examine next. When we reach any node in the tree, we look at the sample clustroids for its children and go next to the child with the clustroid closest to p. Note that some of the sample clustroids at a node may have been seen at a higher level, but each level provides more detail about the clusters lying below, so we see many new sample clustroids each time we go a level down the tree. Finally, we reach a leaf. This leaf has the cluster features for each cluster represented by that leaf, and we pick the cluster whose clustroid is closest to p.