

Assignment 4

WEEK 08: APACHE SPARK

WEEK 09: FEATURE HASHING AND LSH

Teacher and teaching assistant :

David Kofoed Wind,
Finn Årup Nielsen
(Rasmus) Malthe Jørgensen
Ulf Aslak Jensen

CONTENTS

WEEK 08

Exercise 8.1-----	3
Exercise 8.2-----	6
Exercise 8.3-----	8

WEEK 09

Exercise 9.1-----	16
Exercise 9.2-----	19

WEEK 08: APACHE SPARK

Exercise 8.1

Write a Spark job to count the occurrences of each word in a text file. Document that it works with a small example.

Solution:

First we finished Exercise0, with the Spark jobs working correctly.

Here is a short summary of the solution steps:

1. Load a sample .txt file in the working directory to use.
2. Read the file as a collection of lines using [sc.textFile](#).
3. Count the words in each line using [len\(line.split\)](#), and map it to a new RDD.
4. Add up the word count in each line and we can get the final result.

Here's the code. And we got the correct output as the screenshot shows.

```
# takes an URI for the file and reads it as a collection of lines
distFile = sc.textFile("test.txt")
## -- Here below is the content of test.txt --
## AS YOU LIKE IT
## COUNT IT

# This maps each line to an integer value, creating a new RDD
wordCounts = distFile.map(lambda line: len(line.split()))
print "Here's the word count in each line:"
print wordCounts.collect()
print

# Add up the word count in each line and we get the total word count
totalcount = 0
for i in wordCounts.collect():
    totalcount += i
print "Here's the total word count:"
print totalcount
```

```
Here's the word count in each line:
[4, 2]
```

```
Here's the total word count:
6
```

Exercise 8.2:

Write a Spark job that determines if a graph has an Euler tour (all vertices have even degree) where you can assume that the graph you get is connected.

SOLUTION:

First, we split the file into 5 different files manually, deleting the node and edge counts in the top of the file.

To determine if a graph has an Euler tour, which means all vertices have even degree, we should find the total degree of each node in the graph. For a vertex, the number of head ends adjacent to a vertex is called the indegree of the vertex and the number of tail ends adjacent to a vertex is its outdegree. In this case, if we count the nodes in the source column (first column), we can get their outdegree, and similarly if we count them in the target column, we can get their indegree. For each node, if we add up its indegree and outdegree, we can get its total degree. This problem converts to determining if all the nodes have even degree.

Here is a short summary of the solution steps:

1. Read in the file, and get a form of two columns with the source nodes and target nodes.
2. Get the count of source nodes and target nodes, store the results in two lists.
3. Construct a total node list to construct a total node degree list.
4. Determine if it is an Euler graph by looking at the degrees.

Here are the code with documentation and the result screenshot.

```

from pyspark.sql import SQLContext, Row
sqlContext = SQLContext(sc)

# Load a text file and convert each line to a Row.
lines = sc.textFile("graphs/eulerGraphs1.txt")
parts = lines.map(lambda l: l.split(" "))
edge = parts.map(lambda p: Row(source=int(p[0]), target=int(p[1])))

# Infer the schema, and register the DataFrame as a table.
schemaEdge = sqlContext.inferSchema(edge)
schemaEdge.registerTempTable("edge")

# Get the count of source node and target node in list1 & list2
gdf1 = schemaEdge.groupBy(schemaEdge.source)
list1 = gdf1.agg({"*": "count"}).collect()
gdf2 = schemaEdge.groupBy(schemaEdge.target)
list2 = gdf2.agg({"*": "count"}).collect()

# Construct the total nodelist
nodelist = []
for node1 in list1:
    nodelist.append(node1[0])
for node2 in list2:
    if node2[0] not in nodelist:
        nodelist.append(node2[0])

# Construct the degree list by adding the count in list1 & list2 where
# the target node equals the source node
degreeelist = []
for node1 in list1:
    flag = 0
    for node2 in list2:
        if node2[0] not in nodelist:
            continue
        if node2[0] == node1[0]:
            degreeelist.append([node1[0], node1[1] + node2[1]])
            flag = 1
            nodelist.remove(node1[0])
    if flag == 0:
        degreeelist.append([node1[0], node1[1]])
        nodelist.remove(node1[0])
for node2 in list2:
    if node2[0] not in nodelist:
        continue
    degreeelist.append([node2[0], node2[1]])
    nodelist.remove(node2[0])

# Determine if it is an euler graph by looking at each node's degree
# If each node has an even degree, then it is an euler graph
flag = 0
for item in degreeelist:
    if item[1] % 2 != 0:
        flag += 1
        break
if flag != 0:
    print "This is not an euler graph."
else:
    print "This is an euler graph."

```

This is an euler graph.

And we finally got the result that:

First graph is an Euler graph.

Second graph isn't an Euler graph.

Third graph is an Euler graph.

Fourth graph is an Euler graph.

Fifth graph isn't an Euler graph.

Exercise 3:

You are given a couple of hours of raw WiFi data from my phone:

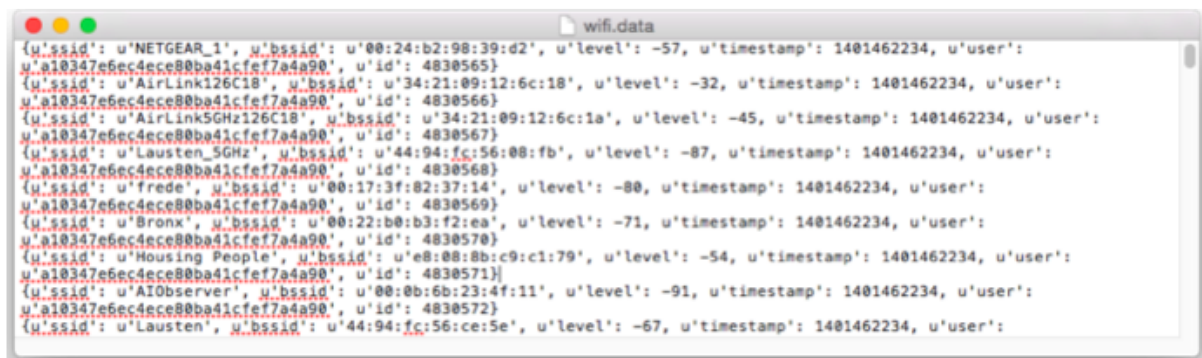
<https://www.dropbox.com/s/964gq5o5bkzg7q3/wifi.data?dl=0>

Compute the following things using Spark:

1. What are the 10 networks I observed the most, and how many times were they observed? Note: the bssid is unique for every network, the name (ssid) of the network is not necessarily unique.
2. What are the 10 most common wifi names? (ssid)
3. What are the 10 longest wifi names? (again, ssid)

SOLUTION:

First we manipulated the .data file to a .txt file, which is more readable for `sc.textFile()` function.



```
{u'ssid': u'NETGEAR_1', u'bssid': u'00:24:b2:98:39:d2', u'level': -57, u'timestamp': 1401462234, u'user': u'a10347e6ec4ece80ba41cfef7a4a90', u'id': 4830565}
{u'ssid': u'AirLink126C18', u'bssid': u'34:21:09:12:6c:18', u'level': -32, u'timestamp': 1401462234, u'user': u'a10347e6ec4ece80ba41cfef7a4a90', u'id': 4830566}
{u'ssid': u'AirLink5GHz126C18', u'bssid': u'34:21:09:12:6c:1a', u'level': -45, u'timestamp': 1401462234, u'user': u'a10347e6ec4ece80ba41cfef7a4a90', u'id': 4830567}
{u'ssid': u'Lausten_5GHz', u'bssid': u'44:94:fc:56:08:fb', u'level': -87, u'timestamp': 1401462234, u'user': u'a10347e6ec4ece80ba41cfef7a4a90', u'id': 4830568}
{u'ssid': u'frede', u'bssid': u'00:17:3f:82:37:14', u'level': -80, u'timestamp': 1401462234, u'user': u'a10347e6ec4ece80ba41cfef7a4a90', u'id': 4830569}
{u'ssid': u'Bronx', u'bssid': u'00:22:b0:b3:f2:ea', u'level': -71, u'timestamp': 1401462234, u'user': u'a10347e6ec4ece80ba41cfef7a4a90', u'id': 4830570}
{u'ssid': u'Housing People', u'bssid': u'e8:08:8b:c9:c1:79', u'level': -54, u'timestamp': 1401462234, u'user': u'a10347e6ec4ece80ba41cfef7a4a90', u'id': 4830571}
{u'ssid': u'AIObserver', u'bssid': u'00:0b:6b:23:4f:11', u'level': -91, u'timestamp': 1401462234, u'user': u'a10347e6ec4ece80ba41cfef7a4a90', u'id': 4830572}
{u'ssid': u'Lausten', u'bssid': u'44:94:fc:56:ce:5e', u'level': -67, u'timestamp': 1401462234, u'user': u'a10347e6ec4ece80ba41cfef7a4a90', u'id': 4830575}
```

Figure 8.3.1 the original wifi.data



```
1 u'NETGEAR_1,u'00:24:b2:98:39:d2,-5,1401462234,u'a10347e6ec4ece80ba41cfef7a4a90,4830565,
2 u'AirLink126C18,u'34:21:09:12:6c:18,-3,1401462234,u'a10347e6ec4ece80ba41cfef7a4a90,4830566,
3 u'AirLink5GHz126C18,u'34:21:09:12:6c:1a,-4,1401462234,u'a10347e6ec4ece80ba41cfef7a4a90,4830567,
4 u'Lausten_5GHz,u'44:94:fc:56:08:fb,-8,1401462234,u'a10347e6ec4ece80ba41cfef7a4a90,4830568,
5 u'frede,u'00:17:3f:82:37:14,-7,1401462234,u'a10347e6ec4ece80ba41cfef7a4a90,4830569,
6 u'Bronx,u'00:22:b0:b3:f2:ea,-7,1401462234,u'a10347e6ec4ece80ba41cfef7a4a90,4830570,
7 u'Housing People,u'e8:08:8b:c9:c1:79,-5,1401462234,u'a10347e6ec4ece80ba41cfef7a4a90,4830571,
8 u'AIObserver,u'00:0b:6b:23:4f:11,-9,1401462234,u'a10347e6ec4ece80ba41cfef7a4a90,4830572,
9 u'Lausten,u'44:94:fc:56:ce:5e,-6,1401462234,u'a10347e6ec4ece80ba41cfef7a4a90,4830573,
10 u'Bismarck,u'5c:d9:98:63:bf:e2,-8,1401462234,u'a10347e6ec4ece80ba41cfef7a4a90,4830574,
11 u'NETGEAR90,u'84:1b:5e:6f:ff:b9,-8,1401462234,u'a10347e6ec4ece80ba41cfef7a4a90,4830575,
```

Figure 8.3.2 the converted wifi.txt

Here's the code with documentation for this.


```
import io

text = []
# Read the file
with open("wifi.data", 'r') as f:
    line = f.read().splitlines()
    for comp in line:
        # Split each element
        item = comp.split(", ")
        for compo in item:
            # Delete the last '}'
            compo = compo[:-1]
            # Ignore the tags like 'ssid'..
            item2 = compo.split(": ")
            text.append(item2[1])
            # Append a split symbol
            text.append(", ")
        text.append("\n")

# print text to a new file
f2 = open('wifi.txt', 'w')
for item in text:
    f2.write("%s" % item)
f2.close()
```

Now we come to the 3 problems!

- I. We want to find the 10 networks he observed the most, and how many times they were observed. As the bssid is unique for every network, the name (ssid) of the network is not necessarily unique, so after constructing the dataframe, we can count the bssid to get the networks he observed the most.

Here is a short summary of the solution steps:

- 1) Load the text file, convert it, infer the schema and register the DataFrame as a table.
- 2) Count each bssid's frequency, and get a new dataframe, and store the top 10 to the list "most".
- 3) Print the result to a table.

Here are the code with documentation and the result screenshot.

```
# Load a text file and convert each line to a Row.
lines = sc.textFile("wifi.txt")
parts = lines.map(lambda l: l.split(","))
wifi = parts.map(lambda p: Row(ssid=p[0], bssid=p[1], \
                               level=p[2], timestamp=p[3], user=p[4], id=p[5]))

# Infer the schema, and register the DataFrame as a table.
schemaWifi = sqlContext.inferSchema(wifi)
schemaWifi.registerTempTable("wifi")

from pyspark.sql.functions import col
from prettytable import PrettyTable

# Count every bssid and get the top 10
most = schemaWifi.groupBy("bssid").count().orderBy(col("count").desc()).take(10)

# Print out the results in a table
t = PrettyTable(['bssid', 'count'])
for row in most:
    t.add_row(row)
print t
```

bssid	count
u'34:21:09:12:6c:1a	347
u'00:24:b2:98:39:d2	338
u'34:21:09:12:6c:18	324
u'e8:08:8b:c9:c1:79	318
u'44:94:fc:56:08:fb	315
u'00:22:b0:b3:f2:ea	314
u'2c:b0:5d:ef:08:2b	272
u'44:94:fc:56:ce:5e	240
u'28:cf:e9:84:a1:c3	211
u'bc:ee:7b:55:1a:43	210

II. We want to find the 10 most common wifi names(ssid).

To do this, we first count the distinct bssid group by ssid. And after sorting the results, we can get the 10 most common wifi names.

Here's the code with documentation, and a result table is attached.

```
import pyspark.sql.functions as func

# for each ssid count the distinct bssid
# so we can get the frequency of each wifi name used for different networks
mostcommon = schemaWifi.groupBy("ssid").agg(col("ssid"), \
    func.countDistinct('bssid')). \
    orderBy(col("COUNT(DISTINCT bssid)").desc()).take(10)

# Print the result to a table
t = PrettyTable(['ssid', 'count'])
for row in mostcommon:
    t.add_row(row)
print t
```

ssid	count
u'	15
u'SIT-GUEST	15
u'SIT-PROD	13
u'SIT-BYOD	13
u'MED-105	11
u'KNS-105	11
u'PDA-105	11
u'KEA-PUBLIC	6
u'GST-105	5
u'wireless	4

III. We want to find the 10 longest wifi names (ssid).

Here is a short summary of the solution steps:

- 1) Map the ssid and the length of each ssid.
- 2) Sort the map by value of the length, and get the top 10 results.
- 3) Print the result to a table.

Here's the code with documentation, and a result table is attached.

```
# Get the map of each ssid and the corresponding length of the name
namelen = wifi.map(lambda word: (str(word[3]),len(word[3])-2))
# Sort the map by the value of length in descending order
finalnamelen = namelen.distinct().sortBy(lambda word: -word[1]).take(10)

# Print the result to a table
t = PrettyTable(['wifiname', 'length'])
for row in finalnamelen:
    t.add_row(row)
print t
```

wifiname	length
u'HP-Print-43-Deskjet 3520 series	31
u'Charlotte R.s Wi-Fi-netv\xe6rk	30
u'TeliaGatewayA4-B1-E9-2C-9E-CA	29
u'Emil M\xfrkebergs Netv\xe6rk	29
u'TeliaGateway08-76-FF-85-04-2F	29
u'TeliaGateway9C-97-26-57-15-99	29
u'TeliaGateway08-76-FF-8A-EE-32	29
u'TeliaGateway08-76-FF-46-3E-36	29
u'TeliaGateway08-76-FF-84-FF-8C	29
u'TeliaGateway08-76-FF-9C-E0-82	29

Week 9 FEATURE HASHING AND LSH

Exercise 9.1

Download all the following files:

<https://github.com/fergiemcdowall/reuters-21578-json/tree/master/data/full>

Load them into Python. Remove all articles that do not have at least one topic and a body. Make a bag-of-words encoding of the body of the articles. Remember to lower-case all words. How many features/columns do you get in your term/document matrix from your bag-of-words encoding?

Train a random forest classifier to predict if an article has the topic 'earn' or not from the body-text (encoded using bag-of-words). How does the classifier perform (how large a fraction of the documents in the test set are classified correctly)?

Now implement feature hashing and use 1000 buckets instead of the raw bag-of-words encoding. How does this affect your classifier performance?

1. Load the files

```
import json
import glob
lst=glob.glob("E:/repo/bigwork2016/week9/data/*.txt")
list_article=[]
list_topic=[]
for i in lst:
    with open(i) as file_i:
        content_string = file_i.read()
        list_dict_json = json.loads(content_string)
        file_i.close()
        for j in list_dict_json:
            if 'topics' in j.keys():
                if 'body' in j.keys():
                    text=j['body'].lower()
                    text=text.replace('\n',' ')
                    list_article.append(text)
                    if 'earn' in j['topics']:
                        list_topic.append(1)
                    else:
                        list_topic.append(0)
print ("Length of list_article is %d" %len(list_article))
```

In this part, we load json-files into python and extract all the articles. Besides, we need judge if an article has the topic 'earn' or not. I think code in this part doesn't need too much explanation because it is similar to that of Exercise 3 of Week 2.

Maybe the only thing needs special attention is that we should replace all the EOF '\n' by space. In the end let's check the number of article. (Of course it is 10377)

2. Extract features

```
list_bag_word=[]
for i in list_article:
    word_list=[]
    uniq_word=[]
    uniq_word=set(i.split(' '))
    word_list.append(sorted(uniq_word))
    list_bag_word.extend(word_list)
dic=[]
for i in list_bag_word:
    dic.extend(i)
dic=list(set(dic))
dic.sort()
print ("Length of dic is %d" %len(dic))
```

After loading files, we need to extract all of the features/words to form the bag of words. The code in this part is also very clear and easy. The main idea is to collect all words from each article and then use `set()` to remove duplicate ones. Just like Part 1, we check the number of features at the end. (Exactly 70794)

3. Form the matrix and simplify the model

```
import random
random_column=random.sample(range(len(dic)),10000)
result=[[0 for col in range(10000)] for row in range(len(list_article))]
for i in range(len(list_bag_word)):
    for j in range(len(random_column)):
        if dic[(random_column[j])] in list_bag_word[i]:
            result[i][j]=1
```

As is shown, the purpose of this part is to form the matrix we want. Maybe you have noticed that number of features is reduced. I have to say that is not necessary, or even no good. The main reason for that is handling such a huge matrix is beyond the capacity of my computer, which leads to loss of data. (Fortunately, the result shows that the prediction model based on it still has a high accuracy.)

4. Train and predict by bag-of-words

```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test
=train_test_split(result,list_topic,test_size=0.2)
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(n_estimators=50)
clf = clf.fit(X_train, y_train)
pred=list(clf.predict(X_test))
```



```
count=0
for i in range(len(y_test)):
    if pred[i]==y_test[i]:
        count+=1
prob=count/len(y_test)
print ("Probability of correct prediction by bag-of word is %f" %prob)
```

First we use *train_test_split()* to divide dataset to training one and the testing one at the rate of 0.2. Then we draw in a *RandomForestClassifier()* and make it fit to the training set. After that the trained classifier is used to predict the result of testing set. Of course we need to how large a fraction of the documents in the test set are classified correctly.

5. Train and predict by feature hashing

```
from sklearn.feature_extraction.text import HashingVectorizer
hv = HashingVectorizer(n_features=1000)
hashing_vector=hv.transform(list_article)
X_train2, X_test2, y_train2, y_test2 =
train_test_split(hashing_vector,list_topic,test_size=0.2)
clf2 = RandomForestClassifier(n_estimators=50)
clf2 = clf2.fit(X_train2, y_train2)
pred2=list(clf2.predict(X_test2))
count2=0
for i in range(len(y_test2)):
    if pred2[i]==y_test2[i]:
        count2+=1
prob2=count2/len(y_test2)
print ("Probability of correct prediction by hash is %f" %prob2)
```

For feature hashing, we need to import a *HashingVectorizer()*. Given the number of features to be 1000, the vectorizer gives out a matrix of size 10377 x 1000. Then comes the split of dataset, the training of classifier and the prediction, which is almost the same as Part 4.

Result

Here are the screenshots of the result.

```
Length of list_article is 10377
Length of dic is 70794
Probability of correct prediction by bag-of word is 0.927746
Probability of correct prediction by hash is 0.961464
```

From result we know that the matrix is of size 10377 x 70794, proving that the formation of matrix is correct. Apart from that, both methods of prediction are highly exact, while featuring hashing is more accurate. We think it is not surprising since the bag-of-words method has lost some data. And we have to say that feature hashing is amazingly accurate and of a high efficiency.

Exercise 9.2

Implement your own MinHash algorithm.

Using the same dataset as before, hash the body of some of the articles (encoded using bag of words) using MinHash – to get the code to run faster, work with just 100 articles to begin with.

Try with different number of hash functions/permutations (for example 3, 5, 10).

Look at which documents end up in the same buckets. Do they look similar? Do they share the same topics?

SOLUTION:

Based on previous exercise, we continue to MinHash based on the first 100 articles we got.

Here is a short summary of the solution steps:

- 1) Create a 0-1 matrix (list of lists), representing if a word appears in the article.
- 2) Do random permutation, and get the correspond new 0-1 vector.
- 3) Find where the first 1 is in each article's sublist, and get the MinHash.
- 4) After doing 3/5/10 random permutations, we can construct the bucket for each article.
- 5) Finally we group articles which have the same bucket, and do analysis.

Here's the code with documentation, and the result is attached.

```
import numpy as np
import random

# Construct a list of lists, with 100 sublists representing 100 articles
# We consider if a word in the unique-word-bag is in the article
# The sequence of the word is as that in the unique-word-bag
# In each sublist, the number 1 denotes that the word is in the article,
# while 0 denotes that word is not in the article

count = []
# Work with 100 articles
for i in range(100):
    temp = []
    for word in dic:
        if word in list_article[i]:
            temp.append(1)
        else:
            temp.append(0)
    count.append(temp)
```



```
# We give an index to each unique word for easier representation
wordlist = range(70794)
# vec stores the 10 MinHash we get
vec = []

for ii in range(10):
    # Perform a random permutation
    random.shuffle(wordlist)
    # Get the permuted 0-1 vector
    x = []
    for j in range(100):
        temp = []
        for i in wordlist:
            temp.append(count[j][i])
        x.append(temp)
    # Find where the first 1 occurs in each article's 0-1 vector
    temp2 = []
    for n in range(100):
        for m in range(len(x[n])):
            if x[n][m] == 1:
                temp2.append(m+1)
                break
    vec.append(temp2)
```

```
# Get the bucket vectors of each article which is stored in a list of lists
bucket = []
for j in range(100):
    temp = []
    for i in range(10):
        temp.append(vec[i][j])
    bucket.append(temp)
```

```
# Group all the articles which have common bucket together
common = []
flag = 0
for i in range(len(bucket)):
    temp = []
    for j in range(len(bucket)):
        if bucket[i] == bucket[j]:
            temp.append(j)
            flag += 1
    common.append(temp)
    if flag == 100:
        break
```

Result Analysis

After hashing the body of the articles, some documents do end up in the same buckets. When number of hash functions is respectively 3, 5 and 10, results are as follows.

```
>>> result_3
[[0], [1], [2], [3, 82], [4], [5], [6], [7, 17], [8, 99], [9], [10], [11], [12],
 [13], [14, 18, 44, 95], [15], [16, 26, 50, 54], [7, 17], [14, 18, 44, 95], [19],
 [20, 74, 90], [21], [22], [23], [24], [25], [16, 26, 50, 54], [27], [28], [29,
 75], [30], [31, 51, 66, 72, 86], [32], [33], [34], [35], [36], [37], [38], [39,
 57], [40], [41], [42], [43], [14, 18, 44, 95], [45], [46, 58, 77, 91], [47, 49,
 78, 84, 89, 92], [48], [47, 49, 78, 84, 89, 92], [16, 26, 50, 54], [31, 51, 66,
 72, 86], [52]]

>>> result_5
[[0], [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14],
 [15, 64], [16, 29, 49], [17], [18], [19], [20], [21], [22], [23], [24], [25], [
 26], [27], [28], [16, 29, 49], [30], [31], [32], [33], [34], [35], [36], [37], [
 38], [39, 58, 78, 84], [40, 86], [41], [42], [43], [44], [45], [46], [47], [48],
 [16, 29, 49], [50], [51], [52], [53], [54], [55], [56], [57], [39, 58, 78, 84],
 [59], [60], [61], [62], [63], [15, 64], [65], [66], [67], [68], [69], [70], [71],
 [72], [73], [74], [75], [76], [77], [39, 58, 78, 84], [79], [80], [81]]

>>> result_10
[[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], [26], [27], [
 28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41],
 [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54],
 [55], [56], [57], [58], [59], [60], [61], [62], [63], [64], [65], [66], [67], [
 68], [69], [70], [71], [72], [73], [74], [75], [76], [77], [78], [79], [80], [81],
 [82], [83], [84], [85], [86], [87], [88], [89], [90], [91], [92], [93], [94],
 [95], [96], [97], [98], [99]]
```

As you can see, number of buckets are respectively 53 and 82 and 100. Hence when there are 3 hash functions nearly a half of articles is in a cluster. As the number of hash functions increase, some documents become outliers and no clusters exist in the end when 10 hash functions come.

Now what we are concerned is if articles in the same cluster has the same or similar topics. Assume that we have a list called *list_of_topic*, which contains the topics of all 100 documents. Then we set up a filter to leave those in a cluster so that we can extract and compare their topics. The code is as follows. (for 3 hash function)

```
result_3_filter=[]
for i in result_3:
    if(len(i)>1):
        result_3_filter.append(i)
comparison=[]
for i in result_3_filter:
    temp=[]
    for j in i:
        temp.append(list_of_topic[j])
    comparison.append(temp)
```

```
for i in comparison:
    print(i)
```

And the result is like this.

```
[['earn'], ['earn']]
[['earn'], ['earn']]
[['earn'], ['earn']]
[['earn'], ['earn'], ['earn'], ['earn']]
[['earn'], ['reserves'], ['earn'], ['money-supply']]
[['earn'], ['earn']]
[['earn'], ['earn'], ['earn'], ['earn']]
[['earn'], ['earn'], ['earn']]
[['earn'], ['reserves'], ['earn'], ['money-supply']]
[['earn'], ['earn']]
[['grain', 'corn'], ['acq'], ['acq'], ['acq'], ['acq']]
[['earn'], ['grain', 'ship']]
[['earn'], ['earn'], ['earn'], ['earn']]
[['earn'], ['earn'], ['earn'], ['earn']]
[['earn'], ['earn'], ['earn'], ['earn'], ['earn'], ['earn']]
[['earn'], ['earn'], ['earn'], ['earn'], ['earn'], ['earn']]
[['earn'], ['reserves'], ['earn'], ['money-supply']]
[['grain', 'corn'], ['acq'], ['acq'], ['acq'], ['acq']]
```

As it is shown, the result is quite good. Most clusters do have articles with same topics in it. And different topics in a group are of similarities, just like 'earn' and 'money-supply'.

When there are 5 hash function, the situation is similar. This time, the number of clusters decreases but the result is more accurate. Only three topics, 'earn', 'acq' and 'housing' appear and most clusters have only one topic.

```
[['housing'], ['earn']]
[['earn'], ['earn'], ['earn']]
[['earn'], ['earn'], ['earn']]
[['earn'], ['earn'], ['earn'], ['earn']]
[['earn'], ['acq']]
[['earn'], ['earn'], ['earn']]
[['earn'], ['earn'], ['earn'], ['earn']]
[['housing'], ['earn']]
[['earn'], ['earn'], ['earn'], ['earn']]
```

As regards the result when 10 hash functions exist, it is not surprising to see there are no clusters. That's because even if words in similar articles are mainly different. Few documents can have same mapping value after 10 random hash functions.