Assignment 3
02807 Computational Tools for Big Data

Anonymous authors

November 30th, 2015

Exercise 8.1

The task is to implement a MapReduce program that counts the number of occurrences of each word in a text file.

We define a mapping that first cleans the input (whitespace, lower, etc), then for each word returns the tuple

$$(\text{Key}, \text{Value}) = (\text{Word}, 1).$$

Then we simply define a reducer that sums up the values for each word.

Input: "Hi all. This is a text file. It is used for counting words such as file, text and all. I hope it works!"

Output:

```
"a" 1                           "i" 1
"all" 2                         "is" 2
"and" 1                         "it" 2
"as" 1                          "such" 1
"counting" 1                    "text" 2
"file" 2                        "this" 1
"for" 1                         "used" 1
"hi" 1                          "words" 1
"hope" 1                        "works" 1
```

In the shown example one can see that it correctly counts the number of occurrences of each word.

Code:

```
from mrjob.job import MRJob
import re

class MRWordFrequencyCount(MRJob):

    # Splits the line into separate words
    def mapper(self, _, line):
        for word in re.sub(r'[^\w]', ' ', line).strip().
            lower().split(' '):
             if len(word)>0:
                 yield word, 1

    # Counts the number of occurences for each word
    def reducer(self, key, values):
        yield key, sum(values)


if __name__ == '__main__':
    MRWordFrequencyCount.run()
```

Exercise 8.2

The task is to implement a MapReduce program that determines whether a connected graph has an Euler tour. This is equivalent to checking that all vertices have even degree, so this is what we will check.

This is slightly more challenging and we will therefore use 2 reducers, and hence we first define in which order the various mappings and reducers are done. Namely, `mapper1` $\to$ `reducer1` $\to$ `reducer2`.

We wish to see if each vertex number occurs an even number of times, hence in the mapping, `mapper1`, we yield the tuples

$$(\text{Key}, \text{Value}) = (\text{Vertex Number}, 1).$$

In the first reducer, `reducer1`, we sum up the values for each vertex number. If the number of occurrences is even we return the number 1, otherwise 0.

In the last reducer, `reducer2`, we use a little trick. We know that there exists an Euler tour if and only if *all* numbers occur an even number of times, that is if we returned only 1s in the previous reducer, otherwise there exists at least one 0. Therefore we simply return the minimum of all the truth values returned in the last reducer. If we in the end return 1 there exists an Euler tour, otherwise we return 0.

Inputs: We test the program on the 5 given graph examples, for example the first input is

```
0 1
1 2
2 3
3 0
```

Outputs: For the 5 examples we get the outputs:

```
"Euler tour: " 1
"Euler tour: " 0
"Euler tour: " 1
"Euler tour: " 1
"Euler tour: " 0
```

Hence regarding the question whether there exists an Euler tour, the answers are **yes**, **no**, **yes**, **yes**, and **no**.

Code:

```
from mrjob.step import MRStep
from mrjob.job import MRJob
import re

class MRWordFrequencyCount(MRJob):

    # Defined the steps for mapreduce
    def steps(self):
        return [MRStep(mapper=self.mapper1,reducer=self.
            reducer1),
            MRStep(reducer=self.reducer2)]

    # Yield each vertex from an edge
    def mapper1(self, _, line):
        for number in line.split(' '):
            if len(number)>0:
                yield number, 1

    # Check if a vertice occurs an even number of times
    def reducer1(self, key, values):
        euler = 0
        if(sum(values) % 2 == 0):
            euler = 1
        yield None, euler

    # Check if all vertices have even degree
    def reducer2(self, _,values):
        yield "Euler tour: ", min(values)

if __name__ == '__main__':
    MRWordFrequencyCount.run()
```

Exercise 8.3

The task is to implement a MapReduce program that finds common friends given a graph structure like the previous exercise.

To solve this problem, we used both 2 mappings and reducers. The first mapper and reducer together perform the very simple task of creating a tuple for each person consisting of the person number and a list of its friends:

$$(\text{Key}, \text{Value}) = (\text{Person Number}, \text{Friend list})$$

The second mapping, `mapper2`, takes these tuples and for each link between persons P_1 and P_2 create two new tuples

$$(\text{Key}, \text{Value}) = ([P_1, P_2], \text{Friend list of } P_1)$$
$$(\text{Key}, \text{Value}) = ([P_1, P_2], \text{Friend list of } P_2)$$

and in the last reducer, `reducer2`, we take the intersection between the friend lists such for each friendship we return

$$([P_1, P_2], [\text{Friend list of } P_1] \cap [\text{Friend list of } P_2])\,.$$

Inputs: The example from the slides is given by

```
A B                     B C                     C D
A C                     B D                     C E
A D                     B E                     D E
```

and we also have the large text file `facebook_combined.txt`.

Outputs: The output from the example from the slides is given by

```
["A", "B"] ["C", "D"]            ["B", "E"] ["C", "D"]
["A", "C"] ["B", "D"]            ["C", "D"] ["A", "B", "E"]
["A", "D"] ["B", "C"]            ["C", "E"] ["B", "D"]
["B", "C"] ["A", "D", "E"]       ["D", "E"] ["B", "C"]
["B", "D"] ["A", "C", "E"]
```

The result should be interpreted as, for example, that person A and B have mutual friends C and D, and similarly for the other lines.

A few lines of the output from the large file are given by

```
["0", "1"] ["119", "126", "133", "194", "236", "280", "299", "315",...
     ... "322", "346", "48", "53", "54", "73", "88", "92"]
["0", "10"] ["142", "169", "200", "277", "285", "291", "323", "332", "67"]
["0", "100"] ["119", "150", "163", "189", "217", "269", "323", "64"]
["0", "101"] ["180", "187", "194", "204", "24", "242", "249", "254",...
    ... "266", "299", "302", "317", "330", "346", "53", "80", "92", "94"]
["0", "102"] ["175", "227", "263", "296", "99"]
```

Code:

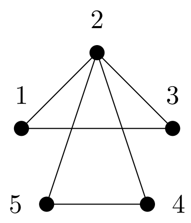
```
from mrjob.job import MRJob
from mrjob.step import MRStep
#import re

class MRWordFrequencyCount(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper1, reducer=self.reducer
                1),
            MRStep(mapper=self.mapper2,reducer=self.reducer2
                )]

    # Split lines
    def mapper1(self, _, line):
        numbers = line.split(' ')
        yield int(numbers[0]), int(numbers[1])

    def reducer1(self, key, values):
        yield key, values

    # Create tuple
    def mapper2(self,key,values):
        for value in values:
            yield (min(key,value),max(key,value)), values

    # Find intersecting set
    def reducer2(self, key,values):
        yield key, list(set(values[0]) & set(values[1]))

if __name__ == '__main__':
    MRWordFrequencyCount.run()
```

Exercise 8.4

The tasks here is to count the number of triangles given a list of vertices between nodes. 2 different ways have been implemented to find the actual triangles. Each consists of 2 sets of a mapper and reducer and then a final reducer.
The first mapper and reducer creates a list of vertices that each vertex is connected to and is the same for both methods.
Then the first method utilizes the same theory as for findings common friends in exercise 8.3. The number of triangles is divided by 3 in the last reducer to make up for the fact that each triangle is counted 3 times.
In the second method we use `itertools.combinations` to construct all possible triangles that each vertex is in considering its connections. Then we check that each triangle has been constructed 3 times meaning that all needed connections are present. Finally the number of triangles are counted.
A small test set has been made, which can be seen below along with its graph. As expected the code returns `Triangles: 2` for both methods.

```
1 2
1 3
2 3
2 4
2 5
4 5
```

(a) Edges in the graph



(b) Corresponding graph

For the large file both methods return `Triangles: 120676`, which corresponds to the result given on their webpage. The first method runs through the large file in 749 seconds, while the second method runs in 854 seconds.

```python
from mrjob.step import MRStep
from mrjob.job import MRJob

import time
import itertools

class MRTriangleCount(MRJob):

    def steps(self):
        return [MRStep(mapper=self.mapper1,reducer=self.
            reducer1),
                MRStep(mapper=self.mapper21,reducer=self.
                    reducer21),
                MRStep(reducer=self.reducer31)]

    # Yield both combinations of vertices from an edge
    def mapper1(self, _, line):
        numbers = line.split(' ')
```

```
            yield int(numbers[0]), int(numbers[1])
            yield int(numbers[1]), int(numbers[0])

    # Collect all vertices that a given vertices is
        connected to
    def reducer1(self, key, values):
        yield key, sorted(list(values))

    # Method 1
    # Construct possible triangles
    def mapper21(self,key, values):
        values = list(values)
        for i in values:
            newKey1 = min(key,i)
            newKey2 = max(key,i)
            yield (newKey1,newKey2), values

    # Check if there is enough connections to make a
        triangle
    def reducer21(self,key,values):
        values = list(values)
        triangles = set(values[0]).intersection(values[1])
        numTriangles = len(triangles)
        yield None, numTriangles

    # Count the total number of triangles
    def reducer31(self, _,values):
        yield "Triangles: ", sum(values)/3

    # METHOD 2
    # Construct possible triangles
    def mapper22(self,key, values):
        for combis in itertools.combinations(values,2):
            newKey = sorted([key, combis[0],combis[1]])
            yield newKey, 1

    # Check if there is enough connections to make a
        triangle
    def reducer22(self,key,values):
        yield None, sum(values) > 2

    # Count the total number of triangles
    def reducer32(self, _,values):
        yield "Triangles: ", sum(values)

if __name__ == '__main__':
    t = time.time()
    MRTriangleCount.run()
    print('Running time: {0:.3f} seconds'.format(time.time()
        -t))
```

Exercise 9.1

In this exercise a Spark job was written to count the occurrences of each word in a text file. We have used the same simple example as was used in exercise 8.1.

The text file is first opened using ”sc.textFile” and saved in lines. We then map the text to words using ”flatMap” because it can return a sequence for each input, here the words, rather than a single output. The words are then mapped to pairs of words and 1, i.e (word,1). The function ”reduceByKey” is then used to sum up the ones by the words, that is counting the words. We then filter out empty words and collect.

Input: ”Hi all. This is a text file. It is used for counting words such as file, text and all. I hope it works!”

Output:

```
[(u'a', 1),
 (u'all', 2),
 (u'used', 1),
 (u'and', 1),
 (u'for', 1),
 (u'this', 1),
 (u'text', 2),
 (u'is', 2),
 (u'it', 2),
 (u'i', 1),
 (u'as', 1),
 (u'hi', 1),
 (u'words', 1),
 (u'such', 1),
 (u'file', 2),
 (u'works', 1),
 (u'counting', 1),
 (u'hope', 1)]
```

We see that we get the same result as in exercise 8.1.

Code:

```
import re

lines=sc.textFile("simple.txt")
words = lines.flatMap(lambda s: re.sub(r'[\W]', ' ', s).
    strip().lower().split(' ') )
pairs = words.map(lambda s: (s,1))
counts = pairs.reduceByKey(lambda a,b: a+b)
counts = counts.filter(lambda x: x[0]!='')
```

```
counts.collect()
```

Exercise 9.2

In this exercise a Spark job that determines if a graph has an Euler tour (all vertices have even degree). This is the same example as was used in exercise 8.2.

We open the text file using "sc.textFile" and save it in lines. We the split the lines into words (the node numbers) using "flatMap", map to pairs and reduce by key to count as we did in 9.1. We the map the the count to either true or false depending on the count being even (true) or odd (false). This is done using modulo 2. We the reduce by taking the minimum of these values, as it will not be an Euler tour if at least one of the counts is odd.

Inputs: We test the program on the 5 given graph examples, for example the first input is

```
0 1
1 2
2 3
3 0
```

Outputs: For the 5 examples we get the outputs:

```
True
False
True
True
False
```

Hence regarding the question whether there exists an Euler tour, the answers are **yes**, **no**, **yes**, **yes**, and **no** which are the same as the answers we got in exercise 8.2.

Code:

```
lines=sc.textFile("graph1.txt")
words = lines.flatMap(lambda s: s.strip().split(' '))
pairs = words.map(lambda s: (s,1))
counts = pairs.reduceByKey(lambda a,b: a+b)
counts.map(lambda x: x[1] \% 2 == 0).reduce(lambda x,y: min(
    x,y))
```

Exercise 9.3

In this exercise we are given a couple of hours of raw WiFi data from a phone and we then have to answer three questions:

- 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.
- What are the 10 most common wifi names? (ssid)
- What are the 10 longest wifi names? (again, ssid)

Inputs: The first five lines in the file looks like this:

```
{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}
```

For **the first question**: 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.

We save each line in the text file in lines using "sc.textFile". Each line is then evaluated using the function "eval" inside a "map" to create a dictionary for each line. We then create pairs of keys and values with the key as (bssid,ssid) and the value 1, i.e. ((bssid,ssid),1). We then use "reduceByKey" to count the number of times each key (wifi) is observed. The pairs are then swapped so we have (count,(bssid,ssid)). This way we can use "sortByKey" with the argument "False" to sort the wifis decreasingly based on their count. Finally, we use "take(10)" to get the top 10.

Output:

```
[(347, (u'34:21:09:12:6c:1a', u'AirLink5GHz126C18')),
 (338, (u'00:24:b2:98:39:d2', u'NETGEAR_1')),
 (324, (u'34:21:09:12:6c:18', u'AirLink126C18')),
 (318, (u'e8:08:8b:c9:c1:79', u'Housing People')),
 (315, (u'44:94:fc:56:08:fb', u'Lausten_5GHz')),
 (314, (u'00:22:b0:b3:f2:ea', u'Bronx')),
 (272, (u'2c:b0:5d:ef:08:2b', u'Playhouse')),
 (240, (u'44:94:fc:56:ce:5e', u'Lausten')),
 (211, (u'28:cf:e9:84:a1:c3', u'Kaspers Wi-Fi-netv\xe6rk')),
 (210, (u'bc:ee:7b:55:1a:43', u'Internet4realz'))]
```

So the most often observed network is called AirLink5GHz126C18, has the bssid 34:21:09:12:6c:1a and is observed 347 times.

Code:

```
lines = sc.textFile("wifi.data")
wifis = lines.map(eval)
pairs = wifis.map(lambda s: ((s["bssid"],s["ssid"]),1))
counts = pairs.reduceByKey(lambda a,b: a+b)
counts = counts.map(lambda s: (s[1],s[0]))
sortc = counts.sortByKey(False)
sortc.take(10)
```

The second question is: What are the 10 most common wifi names? (ssid)

The simple way of understanding this question is just counting the number of times each wifi is observed but based only on ssid. This is done in the same way as the first question except the pairs are (ssid,1) before we reduce by key. This gives the following output and code.

Output:

```
[(402, u'Kaspers Wi-Fi-netv\xe6rk'),
 (402, u'Internet4realz'),
 (347, u'AirLink5GHz126C18'),
 (338, u'NETGEAR_1'),
 (324, u'AirLink126C18'),
 (318, u'Housing People'),
 (315, u'Lausten_5GHz'),
 (314, u'Bronx'),
 (272, u'Playhouse'),
 (240, u'Lausten')]
```

The most common wifi name is then Kaspers Wi-Fi-netv\ xe6rk which was observed 402 times.

Code:

```
lines = sc.textFile("wifi.data")
wifis = lines.map(eval)
pairs = wifis.map(lambda s: (s["ssid"],1))
counts = pairs.reduceByKey(lambda a,b: a+b)
counts = counts.map(lambda s: (s[1],s[0]))
sortc = counts.sortByKey(False)
sortc.take(10)
```

An alternative way to understand the second question is by looking at which names are most common for wifis that are actually different. This we do by again creating the pairs ((bssid,ssid),1) but now we group these by key, so we only have each pair of (bssid,ssid) once. We then create a new pair that is (ssid, 1) from these and count and order in the same way as above.

This gives us most common wifi names for different wifis.

Output:

```
[(15, u''),
 (15, u'SIT-GUEST'),
 (13, u'SIT-BYOD'),
 (13, u'SIT-PROD'),
 (11, u'PDA-105'),
 (11, u'KNS-105'),
 (11, u'MED-105'),
 (6, u'KEA-PUBLIC'),
 (5, u'GST-105'),
 (4, u'wireless')]
```

The two most common names for wifis are an empty name or the name SIT-GUEST which were both observed for 15 different bssid's. It makes sense that institutions will use the same name for different wifis used around the premises and therefore these types of names are many the most common when considering different bssid's.

Code:

```
lines = sc.textFile("wifi.data")
wifis = lines.map(eval)
pairs = wifis.map(lambda s: ((s["bssid"],s["ssid"]),1))
counts = pairs.groupByKey()
counts = counts.map(lambda s: (s[0][1],1))
counts = counts.reduceByKey(lambda a,b: a+b)
counts = counts.map(lambda s: (s[1],s[0]))
sortc = counts.sortByKey(False)
sortc.take(10)
```

The third question is: What are the 10 longest wifi names? (again, ssid)

We first read in the dictionaries using ”eval”. Then we create pairs of (ssid,1) so we can group by key, here ssid. This is done so each name is only there once. We then create new pairs of (length of ssid, ssid), sort decreasingly by key (length of ssid) and take the first 10.

Output:

```
[(31, u'HP-Print-43-Deskjet 3520 series'),
 (29, u'TeliaGatewayA4-B1-E9-2C-9E-CA'),
 (29, u'TeliaGateway08-76-FF-84-FF-8C'),
 (29, u'TeliaGateway9C-97-26-57-15-F9'),
 (29, u'TeliaGateway08-76-FF-46-3E-36'),
 (29, u'TeliaGateway9C-97-26-57-15-99'),
 (29, u'TeliaGateway08-76-FF-8A-EE-32'),
 (29, u'TeliaGateway08-76-FF-85-04-2F'),
 (29, u'TeliaGateway08-76-FF-9C-E0-82'),
 (27, u'Charlotte R.s Wi-Fi-netv\xe6rk')]
```

The longest wifi name is ”HP-Print-43-Deskjet 3520 series” with 31 characters.

Code:

```
lines = sc.textFile("wifi.data")
wifis = lines.map(eval)
pairs = wifis.map(lambda s: (s["ssid"],1))
counts = pairs.groupByKey()
counts = counts.map(lambda s: (len(s[0]),s[0]))
sortc = counts.sortByKey(False)
sortc.take(10)
```

Exercise 10.1 – What is Deep Learning?

Deep learning is a sub discipline of machine learning. It is a set of methods that are used for automatic complex feature extraction in unstructured data. The set of extracted features are not predefined feature types but rather created by the algorithm from the unstructured data, i.e. the methods try to create an alternative representation of the data by analyzing and recognizing certain patterns. This means the computer chooses the best representation of a feature for the computer instead of humans trying to define how it should recognize this feature. It works by using multiple layers of non-linear transformations of the features. Each layer uses the output of the previous layer as the input and this creates a hierarchical representation of the data. The resulting features can then be used for e.g. classification. *(135 words)*

Exercise 10.2 – Convolutional Neural Network

Artificial Neural Networks (ANNs) are a group of models that are used in machine learning whose objective is to estimate functions that depend on a large number of variables. For example classifying what is portrayed in an image, perhaps a cat, given the actual image as the set of variables, i.e. the set of pixels and their positions. Artificial neural networks are inspired by the brain, hence the name.

The closest and simplest analogy of an ANN is perhaps the decision tree. Given a set of variables, we follow a path and each time we reach a node where the path splits into multiple paths, we follow the one determined by a particular rule depending on the set of variables we are given. ANNs are analogous to decision trees in this sense, and rules can automatically be learnt, however the rules are often very difficult to interpret for ANNs. The neurons in ANNs are also much more interconnected than the nodes in decision trees. Neural networks have shown great performance, but because of the because of the previously mentioned facts they can be hard to understand.

A Convolutional Neural Network (CNN) is a particular type of artificial neural network in which we allow for multiple copies of each neuron. They can be seen as analogous to mathematical functions, for example one might have a neuron that attempts to detect certain structures (e.g. edge detecting) in a set of pixels. This neuron could locally be used in many places to detect these structures in an entire image. Using the same neuron many times, instead of many different neurons, can make the convolutional neural network use less memory and also make the model easier to learn – which improves performance. These are key factors when dealing with large data sets. *(299 words)*

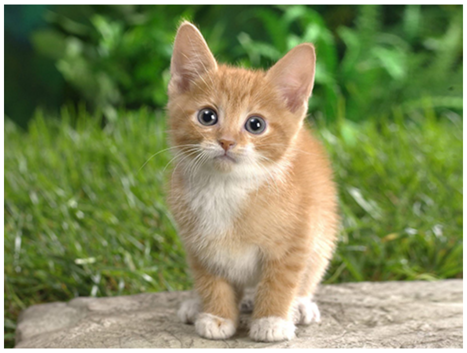
Exercise 10.3 – Caffe Deep Learning

The task in this exercise is to classify what is portrayed in an image using the Caffe Deep Learning framework. We first do the required initial work of downloading the Vagrant file, setting up the virtual machine, and opening up the Python notebook.

First Image

We perform the steps with the default picture which can be seen in Figure 2.

Figure 2: The default picture of a cute cat.



The output of the classification is:

```
Predicted class is #282.
```

The description of this class is "Tiger Cat." In fact, the classifier gives a list of its top guesses:

```
'n02123159 tiger cat'
'n02123045 tabby, tabby cat'
'n02124075 Egyptian cat'
'n02119022 red fox, Vulpes vulpes'
'n02127052 lynx, catamount'
```

We, not being experts in cat breeds, cannot say for sure if this can be considered of the breed "Tiger cat." With that said, we must still say that the performance in this example is quite good as the three top guesses are types of cats. Furthermore, simply being able to classify into various breeds of cats is probably more than the average person can from looking at a picture.

The classification took for us 5.69 seconds:

```
1 loops, best of 3: 5.69 s per loop
```

Second Image

For our own choice of an image, we chose a picture of a peacock which can be seen in Figure 3. We go through the same steps and get the classification:

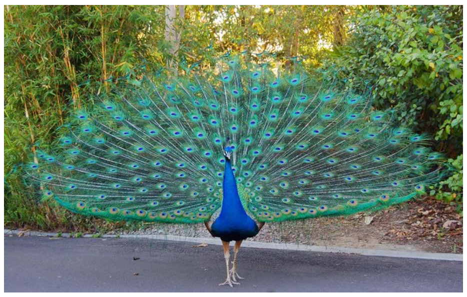
```
Predicted class is #84.
```

This class has the description "Peacock," as desired. We can list the top guesses as before:

```
'n01806143 peacock'
'n01537544 indigo bunting, indigo finch, indigo bird, Passerina cyanea'
'n01580077 jay'
'n02009229 little blue heron, Egretta caerulea'
'n02017213 European gallinule, Porphyrio porphyrio'
```

This time the performance of the classification is obviously very good, simply because the classified class is exactly what is portrayed in the image. Furthermore, the other top guesses are types of blue bird species which are also reasonable guesses.

Figure 3: Our choice of a picture of a peacock.



The classification took for us 5.49 seconds:

```
1 loops, best of 3: 5.49 s per loop
```

Exercise 11.1

We are to create a bag of words based on a number of Reuters articles and then train a random forest classifier using the bag of words matrix. Then we should train a random forest classifier using feature extraction.
For the implementation scipys implementation of sparse matrices are used. A lil-matrix is used for the manipulation and insertion, which is then changed to a csr-matrix, because csr-matrices provide indexing, but is bad at manipulation.
The class `bagOfWords` implements the classic bag of words. It has 4 functions including its initialisation. The function `add_article_to_dict` adds all the words from an article to a dictionary, which includes an index for each word. The dictionary is created first for performance. The function `add_article_to_matrix` adds an article to the bag of words matrix as a row. The function `add_article_topics` builds a dictionary of all the topics including which articles has each topic.
The bag of words matrix which is constructed has 10377 articles and has 43217 features.
The class `featureHashing` implements a bag of words with feature hashing. Since we know the number of features there is no need to build a dictionary of all the words first. It therefore only has 3 functions including the initialisation. The function `add_article_to_matrix` inserts an article into the bag of words by hashes each word into one of the buckets. Similarly to the raw bag of words the function `add_article_topics` construct a dictionary containing all the topics and the articles.
In order to compare the performance of the methods the same test and training set is used. As can be seen below the raw bag of words perform best, but also has the longest running time. For 10000 buckets we have almost the same performance as the raw bag of words. But for lower bag of words the runtime is also lower making it a trade-off between running time and classifier performance.

```
Classifier performance using bag of words: 0.954
Bag of words: 92.550 seconds
Classifier performance using feature hashing with 10 buckets: 0.879
Feature hashing runtime: 58.40 seconds
Classifier performance using feature hashing with 100 buckets: 0.925
Feature hashing runtime: 81.41 seconds
Classifier performance using feature hashing with 1000 buckets: 0.946
Feature hashing runtime: 83.54 seconds
```

```
import json
import os
import re

import scipy.sparse as sps
import numpy as np
from sklearn.ensemble import RandomForestClassifier
import mmh3
import time

class bagOfWords:

    def __init__(self):

        self.dictionary = {}
        self.article_topics = {}
        self.num_articles = 0

    def add_article_to_dict(self, article):
        body = article['body']
        words = re.sub("[^\w'.]+",' ',body).lower().strip().
            split(' ')
        for word in words:
            if (word not in self.dictionary) and (word is
                not ''):
                 self.dictionary[word] = len(self.dictionary)
        self.n = len(self.dictionary)


    def add_article_to_matrix(self,article):


        body = article['body']
        words = re.sub("[^\w'.]+",' ',body).lower().strip().
            split(' ')

        article_array = sps.lil_matrix((1,self.n))

        for word in words:
            _idx = self.dictionary[word]
            article_array[0,_idx] += 1


        if not hasattr(self,'bow_matrix'):
            self.bow_matrix = article_array.tocsr()
        else:
            self.bow_matrix = sps.vstack([self.bow_matrix,
                article_array.tocsr()])


    def add_article_topics(self,article):
        for topic in article['topics']:
            if topic not in self.article_topics:
                 self.article_topics[topic] = []
            self.article_topics[topic].append(self.
                num_articles)
```

```
class featureHashing:

    def __init__(self,buckets):

        self.buckets = buckets
        self.article_topics = {}
        self.num_articles = 0


    def add_article_to_matrix(self,article):


        body = article['body']
        words = re.sub("[^\w'.]+",' ',body).lower().strip().
            split(' ')

        article_array = sps.lil_matrix((1,self.buckets))

        for word in words:
            _idx = mmh3.hash(word) % self.buckets
            article_array[0,_idx] += 1

        if not hasattr(self,'bow_matrix'):
            self.bow_matrix = article_array.tocsr()
        else:
            self.bow_matrix = sps.vstack([self.bow_matrix,
                article_array.tocsr()])


    def add_article_topics(self,article):
        for topic in article['topics']:
            if topic not in self.article_topics:
                self.article_topics[topic] = []
            self.article_topics[topic].append(self.
                num_articles)

folder = 'data'
files = os.listdir(folder)


# ---------------------------------------------------
#                 Bag of words
# ---------------------------------------------------

t = time.time()

bow = bagOfWords()

# Add the words from the articles to the word dictionary
for file in files:
    data = open(os.path.join(folder,file))
    articles = json.load(data)
    for article in articles:
        if ('body' in article) and ('topics' in article):
            bow.add_article_to_dict(article)

# Construct the bag of words matrix and topic dictionary
for file in files:
    data = open(os.path.join(folder,file))
    articles = json.load(data)
    for article in articles:
        if ('body' in article) and ('topics' in article):
            bow.add_article_to_matrix(article)
            bow.add_article_topics(article)
            bow.num_articles += 1

# Calculate size of training set and randomly permutate set
number_train = round(0.8*bow.num_articles)
order = np.random.permutation(bow.num_articles)

# Construct target array for the articles
y = np.zeros(bow.num_articles,dtype=bool)
y[bow.article_topics['earn']] = True

# Construct training and test matrices/arrays
X_train = bow.bow_matrix[order[:number_train-1],:]
y_train = y[order[:number_train-1]]
X_test = bow.bow_matrix[order[number_train:],:]
y_test = y[order[number_train:]]

# Train classifier, fit to training data and calculate
    prediction rate
rfc = RandomForestClassifier(50)
rfc.fit(X_train,y_train)
p_bow = sum(rfc.predict(X_test) == y_test)/float(len(y_test)
    )
print('Classifier performance using bag of words: {}'.format
    (p_bow))
print('Bag of words: {0:.3f} seconds'.format(time.time()-t))

# --------------------------------------------------
#                Feature hashing
# --------------------------------------------------
buckets = [10, 100, 1000]
for bucket in buckets:
    t = time.time()
    fh = featureHashing(bucket)
    # Construct the bag of words matrix and topic dictionary
    for file in files:
        data = open(os.path.join(folder,file))
        articles = json.load(data)
        for article in articles:
            if ('body' in article) and ('topics' in article)
                :
                fh.add_article_to_matrix(article)
                fh.add_article_topics(article)
                fh.num_articles += 1

    # Calculate size of training set and randomly permutate
        set
    #number_train = round(0.8*fh.num_articles)
    #order = np.random.permutation(fh.num_articles)

    # Construct target array for the articles
    y = np.zeros(fh.num_articles,dtype=bool)
    y[fh.article_topics['earn']] = True

    # Construct training and test matrices/arrays
    X_train = fh.bow_matrix[order[:number_train-1],:]
    y_train = y[order[:number_train-1]]
    X_test = fh.bow_matrix[order[number_train:],:]
    y_test = y[order[number_train:]]

    # Train classifier, fit to training data and calculate
        prediction rate
    rfc = RandomForestClassifier(50)
    rfc.fit(X_train,y_train)
    p_feature = sum(rfc.predict(X_test) == y_test)/float(len
        (y_test))
    print('Classifier performance using feature hashing with
         {} buckets: {}'.format(bucket, p_feature))
    print('Feature hashing runtime: {0:.2f} seconds'.format(
        time.time()-t))
```

Exercise 11.2

We are to implement the minHash function. For this we reuse the bag of words class from exercise 11.1 to construct the bag of words matrix. This time we also construct a dictionary with all the articles for quick reference. The function for this is called `add_article`.
First we construct the bag of words as before. Then we do the permutations of the feature set and construct the hashed matrix. The last part checks the distances between the new feature vectors to find out if the articles are in the same buckets. It then prints similar articles and their actual jaccard distance.
For the test case we only use 100 articles. Over 10 runs using 3 permutations we get on average 17.7 collisions. The output for one run can be seen below. Most of them have a low jaccard distance but one is 0.88, which is quite high.

```
Bag of words: 0.432 seconds
Locality hashing: 0.223 seconds
Distance check: 0.132 seconds
Jaccard distance - 47 and 92: 0.47
Jaccard distance - 13 and 89: 0.88
Jaccard distance - 46 and 58: 0.40
Jaccard distance - 58 and 84: 0.33
Jaccard distance - 46 and 84: 0.47
Number of similar articles: 5
```

Using 5 permutations we get on average 4.7 collisions per run. Using 10 permutations we get on average 0.4 collisions. The output of one run with 10 permutations can be seen below along with the 2 articles having an collision. As can be seen then the jaccard distance is a lot lower than for 3 permutations and the articles are very similar in their body and have the same topics.

```
Bag of words: 0.438 seconds
Locality hashing: 0.717 seconds
Distance check: 0.147 seconds
Jaccard distance - 39 and 84: 0.27
Number of similar articles: 1

Article 1 - body:
Qtly div eights cts vs eight cts prior
    Pay April one
    Record March 13
 Reuter
Article 1 - topics: [u'earn']

Article 2 - body:
Qtly div three cts vs three cts prior
    Pay April three
    Record March 13
 Reuter
Article 2 - topics: [u'earn']
```

For the entire dataset we get using 3 permutations 55391 collisions. Using 5 permutations we get 2830 collisions. For 10 permutations there are 368 collisions. A quick look through the articles show that the articles, which are similar at 10 permutations are either copies or very short articles like the ones above, which have a low jaccard difference.

```
import json
import os
import re

import scipy.sparse as sps
import scipy as sp
from scipy.spatial.distance import pdist, squareform
import numpy as np

import time

class bagOfWords:

    def __init__(self):

        self.dictionary = {}
        self.article_topics = {}
        self.num_articles = 0
        self.article = {}

    def add_article_to_dict(self, article):
        body = article['body']
        words = re.sub("[^\w'.]+",' ',body).lower().strip().
            split(' ')
        for word in words:
            if (word not in self.dictionary) and (word is
                not ''):
                 self.dictionary[word] = len(self.dictionary)
        self.n = len(self.dictionary)


    def add_article_to_matrix(self,article):


        body = article['body']
        words = re.sub("[^\w'.]+",' ',body).lower().strip().
            split(' ')

        article_array = sps.lil_matrix((1,self.n))

        for word in words:
                _idx = self.dictionary[word]
                article_array[0,_idx] += 1


        if not hasattr(self,'bow_matrix'):
            self.bow_matrix = article_array.tocsr()
        else:
            self.bow_matrix = sps.vstack([self.bow_matrix,
                article_array.tocsr()])


    def add_article_topics(self,article):
        for topic in article['topics']:
            if topic not in self.article_topics:
                self.article_topics[topic] = []
            self.article_topics[topic].append(self.
                num_articles)

    def add_article(self,article):
        n = len(self.article)
        self.article[n] = {}
        self.article[n]['body'] = article['body']
        self.article[n]['topics'] = article['topics']


folder = 'data'
files = os.listdir(folder)


# ---------------------------------------------------
#                 Bag of words
# ---------------------------------------------------

t = time.time()

bow = bagOfWords()

# Add the words from the articles to the word dictionary
#idx = 0
for file in files:
    data = open(os.path.join(folder,file))
    articles = json.load(data)
    for article in articles:
        if ('body' in article) and ('topics' in article):
            bow.add_article_to_dict(article)
#             idx += 1
#         if idx == 100:
#             break
#     if idx == 100:
#         break
# Construct the bag of words matrix and topic dictionary
#idx = 0
for file in files:
    data = open(os.path.join(folder,file))
    articles = json.load(data)
    for article in articles:
        if ('body' in article) and ('topics' in article):
            bow.add_article_to_matrix(article)
            bow.add_article_topics(article)
            bow.add_article(article)
            bow.num_articles += 1
#             idx += 1
#         if idx == 100:
#             break
#     if idx == 100:
#         break


print('Bag of words: {:.3f} seconds'.format(time.time()-t))

# ---------------------------------------------------
#                Locality hashing
# ---------------------------------------------------

t = time.time()
numPerm = 10
num_features = bow.bow_matrix.shape[1]

hashed_matrix = np.zeros((numPerm,bow.num_articles))

for perm in range(numPerm):
    order = np.random.permutation(num_features)
    for article in range(bow.num_articles):
        _idx = sorted(bow.bow_matrix[article,order].nonzero(
            )[1])[0]
        hashed_matrix[perm,article] = _idx
print('Locality hashing: {:.3f} seconds'.format(time.time()-
    t))

# ---------------------------------------------------
#                Check distances
# ---------------------------------------------------
#
t = time.time()
# Get jaccard distance between articles
d_jaccard = pdist(bow.bow_matrix.toarray(),'jaccard')
d_jaccard = squareform(d_jaccard)

# Get similar articles
```

```
d = pdist(hashed_matrix.transpose())
d = squareform(d)
np.fill_diagonal(d,1)
sim = (d == 0).nonzero()

# Get similar articles as a tuple
similar = set()
for i in range(len(sim[0])):
    key1 = min(sim[0][i],sim[1][i])
    key2 = max(sim[0][i],sim[1][i])
    similar.add((key1,key2))
similar = list(similar)
print('Distance check: {:.3f} seconds'.format(time.time()-t)
   )

# Print the jaccard distance of similar distances
for i in range(len(similar)):
    _idx1 = similar[i][0]
    _idx2 = similar[i][1]
    print('Jaccard distance - {} and {}: {:.2f}'.format(_idx
        1,_idx2,d_jaccard[_idx1,_idx2]))
print('Number of similar articles: {}'.format(len(similar)))
```