Data Mining 2015 - Homework 2

Ludovico Fabbri 1197400

April 12, 2015

1 Problem 1

Here we are asking to implement nearest-neighbor search for text documents. You have to implement shingling, minwise hashing, and locality-sensitive hashing.

1.1

Implement a class that, given a document, creates its set of character shingles of some length k. Then represent the document as the set of the hashes of the shingles, for some hash function.

```python
from Helpers import *

class ShingleDocument:
    def __init__(self, doc, shingleSize):
        self.shingles = set()

        if len(doc) <= shingleSize:
            return doc

        # sequences of two or more white spaces will be normalized
    to one white space
        docProcessed = normalizeWhiteSpaces(doc)

        for i in range(0, len(docProcessed) - (shingleSize-1)):
            shingle = docProcessed[i:i+shingleSize]
            hashedShingle = hash(shingle)
            self.shingles.add(hashedShingle)
```

Simple class with an attribute of type set named 'shingles' where are stored the hashes of the shingles for that document object. The built in set in python assures that duplicates of shingles are discarded. Before computing the shingles, there is an helper function from Helper.py that normalizes white spaces in a preprocessing step.

1.2

Implement a class, that given a collection of sets of objects (e.g., strings, or numbers), creates a minwise hashing based signature for each set.

```
from Helpers import *

class MinHashSets:
    def __init__(self, collectionShigleDocs, n):

        # Dictionary (key=index of document, value=signature of the
     document)
        self.minHashSets = {}

        # Generating minhash family functions
        hashFunctions = []
```

```python
        for i in range(n):
            hashFunctions.append(hashFamily(i))

        # LSH alg
        i = 0
        for shingleDoc in collectionShigleDocs:
            j = 0
            for hashFunc in hashFunctions:
                shingles = set()
                for shingle in shingleDoc.shingles:
                    shingles.add(hashFunc(str(shingle)))

                if j == 0:
                    self.minHashSets[i] = [min(shingles)]
                else:
                    self.minHashSets[i].append(min(shingles))
                j = j + 1
            i = i + 1
```

The dictionary 'minHashSets' stores the signature for each document. After generating a family of n minhash functions we start the minhash algorithm generating the signature for each document.

1.3

Implement a class that implements the locally sensitive hashing (LSH) technique, so that, given a collection of minwise hash signatures of a set of documents, it finds the all the documents pairs that are near each other.

```python
class LSH:
    def __init__(self, minHashSets, numBands, bandWidth, n):

        self.bandsArray = []
        self.candidatePairs = []
        self.nearDuplicates = set()

        start = 0
        for b in range(numBands):
            newDic = {}
            self.bandsArray.append(newDic)
            if (b != 0):
                start += bandWidth

            for key in minHashSets:
                signature = minHashSets[key]
                sliceSignature = signature[start:start+bandWidth]
                sliceSignatureConcat = ""

                for minhash in sliceSignature:
                    sliceSignatureConcat += minhash
                    currentDictionary = self.bandsArray[b]
                hashBucketKey = hash(sliceSignatureConcat)

                if not currentDictionary.has_key(hashBucketKey):
                    currentDictionary[hashBucketKey] = []
                    currentDictionary[hashBucketKey].append(key)
                else:
                    for documentKey in currentDictionary[hashBucketKey]:
                        candidatePair = (key,documentKey)
                        self.candidatePairs.append(candidatePair)
                    currentDictionary[hashBucketKey].append(key)

        # remove symmetric pairs
        self.candidatePairs = set((a,b) if a<=b else (b,a) for a,b in self.candidatePairs)

        # select near-duplicates
        for candidatePair in self.candidatePairs:
            duplicateKey = candidatePair[1]
            self.nearDuplicates.add(duplicateKey)
```

In this class there 3 attributes:

- bandsArray: for each band stores a dictionary of the candidate-pairs
- candidatePairs: a list of all the candidate pairs found by the lsh algorithm
- nearDuplicates: a set of all the near-duplicates found by the lsh algorithm

For each band we iterate over signatures and slice each signature to the width of a band (r). We hash this slice-signature to a bucket (the key of the dictionary) and append the document index to the list related to that key. We have to remove symmetric pairs (example (1,2) and (2,1)) before computing near-duplicates documents, which we do in the last step.

1.4

To test the LSH algorithm, also implement a class that given the shingles of each of the docu- ments, finds the nearest neighbors by comparing all the shingle sets with each other.

```
from Helpers import jaccardSim

class ShingleSimilarity:
    def __init__(self,shingleDocuments):

        self.neighboursPairs = set()
        self.nearDuplicates = set()

        for i in range(len(shingleDocuments)):
            shingleDoc1 = shingleDocuments[i]
            for j in range (i+1, len(shingleDocuments)):
                #if (j != i):
                shingleDoc2 = shingleDocuments[j]
                if jaccardSim(shingleDoc1.shingles, shingleDoc2.
    shingles) >= 0.8:
                    pair = (i,j)
                    self.neighboursPairs.add(pair)

        # remove symmetric pairs
        self.neighboursPairs = set((a,b) if a<=b else (b,a) for a,b
     in self.neighboursPairs)

        # select near-duplicates
        for pair in self.neighboursPairs:
            duplicateKey = pair[1]
            self.nearDuplicates.add(duplicateKey)
```

This class simply use an helper method to compute Jaccard similarity between each set of shingles and the others set, selecting only them that have an index of similarity greater than 0.8. There are 2 attributes:

- neighboursPairs: a set of the pairs of the near-duplicates
- nearDuplicates: a set of the near-duplicates

1.5

Here the file that implements the helpers methods:

```
import hashlib
import re

# regular expression for one or more tabs
spacesRE = re.compile("\s+")


# Format a text document converting the sequences of two ore more
    whitespaces in one whitespace
def normalizeWhiteSpaces(doc):
    result = re.sub(spacesRE, " ", doc);
    return result


# Implement a family of hash functions. It hashes strings and takes
     an # integer to define the member of the family.
# Return a hash function parametrized by i
def hashFamily(i):
    resultSize = 8       # how many bytes we want back
    maxLen = 20          # how long can our i be (in decimal)
    salt = str(i).zfill(maxLen)[-maxLen:]
    def hashMember(x):
        hasher = hashlib.sha1(x + salt).digest()[-resultSize:]
        return hasher
    return hashMember


# Jaccard Similarity
def jaccardSim(shingles1, shingles2):
    intersection = len(set(shingles1).intersection(shingles2))
    return intersection / float(len(shingles1) + len(shingles2) -
    intersection)
```

1.6

This is the main program, where there is computational time for each step of the problem, the number of near duplicates found using LSH and Jaccard similarity respectively and the size of the intersection. For the input i am using output file of the first homework (problem6output.tsv) and selecting each long description as the document.

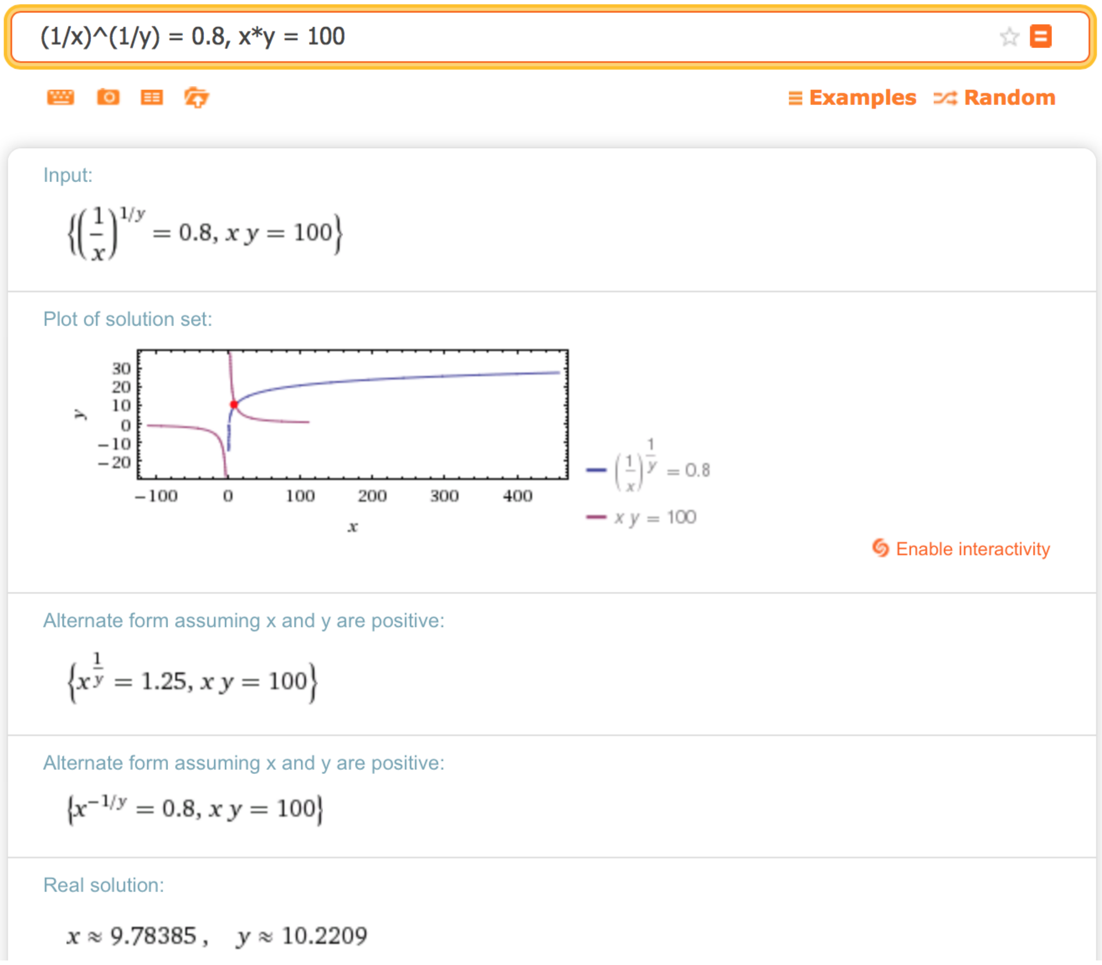
```
#!/usr/bin/env python

from ShingleDoc import ShingleDocument
from MinHash import MinHashSets
from LSH import LSH
from ShingleSimilarity import ShingleSimilarity
import time

print""
print ("START")

k = 10      # shingle size
n = 100     # number of permutations (minhash functions)
b = 10      # number of bands
r = 10      # number of rors for band (bandridth)

print""
print " —————————————————————— "
print "Shingle size: " + str(k)
print "Number of minhash permutations: " + str(n)
print "Number of bands: " + str(b)
print "Number of rors for band: " + str(r)
print " —————————————————————— "
print""


```

```
file = open("problem6output.tsv", 'r')
jobs = file.read().split('\n')
file.close()

shingleDocuments = []
end = len(jobs)-1



print "Computing shingles..."
start = time.time()
for i in range(0, end):
    job = jobs[i]
    jobDescription = job.split('\t')[5]
    shingleDoc = ShingleDocument(jobDescription, k)
    shingleDocuments.append(shingleDoc)
shiglesTime = time.time() - start
print "DONE. Time: " + str(shiglesTime)


print""
print " ------------------------------------------------ "
print""


print "Computing minhashings..."
minhashingTimeStart = time.time()
minHashSets = MinHashSets(shingleDocuments, n).minHashSets
minhashingTimeEnd = time.time() - minhashingTimeStart
print "DONE. Time: " + str(minhashingTimeEnd)
#for key in minHashSets:
#    print len(minHashSets[key]), minHashSets[key]


print""
```

```
print " ————————————————————————— "
print""


print "Computing LSH..."
lshTimeStart = time.time()
lsh = LSH(minHashSets, b, r, n)
lshTimeEnd = time.time() - lshTimeStart
print "DONE. Time: " + str(lshTimeEnd)
candidatePairs = lsh.candidatePairs
nearDuplicatesLSH = lsh.nearDuplicates
print "Number of pairs: " + str(len(candidatePairs))
print "Number of duplicates: " + str(len(nearDuplicatesLSH))


print""
print " ————————————————————————— "
print""


print "Computing Jaccard Similarity..."
jaccartTimeStart = time.time()
shingleSimilarity = ShingleSimilarity(shingleDocuments)
jaccartTimeEnd = time.time() - jaccartTimeStart
print ("DONE. Time: " + str(jaccartTimeEnd))
neighboursPairs = shingleSimilarity.neighboursPairs
nearDuplicatesJS = shingleSimilarity.nearDuplicates
#print shingleSimilarity.neighboursPairs
#print shingleSimilarity.nearDuplicates
print "Number of pairs: " + str(len(neighboursPairs))
print "Number of duplicates: " + str(len(nearDuplicatesJS))


print""
print " ————————————————————————— "
```

```
print""

print "Intersection between Jaccard and LSH near-duplicates: " +
    str(len(set(nearDuplicatesJS).intersection(nearDuplicatesLSH)))


print""
print "END"
```

To find the right values for b and r i've used Wolfram, with n fixed at 100 and the thereshold t=0.8


(1/x)^(1/y) = 0.8, x*y = 100

Input:

$$\left\{\left(\frac{1}{x}\right)^{1/y} = 0.8,\ x\,y = 100\right\}$$

Plot of solution set:

Alternate form assuming x and y are positive:

$$\left\{x^{\frac{1}{y}} = 1.25,\ x\,y = 100\right\}$$

Alternate form assuming x and y are positive:

$$\left\{x^{-1/y} = 0.8,\ x\,y = 100\right\}$$

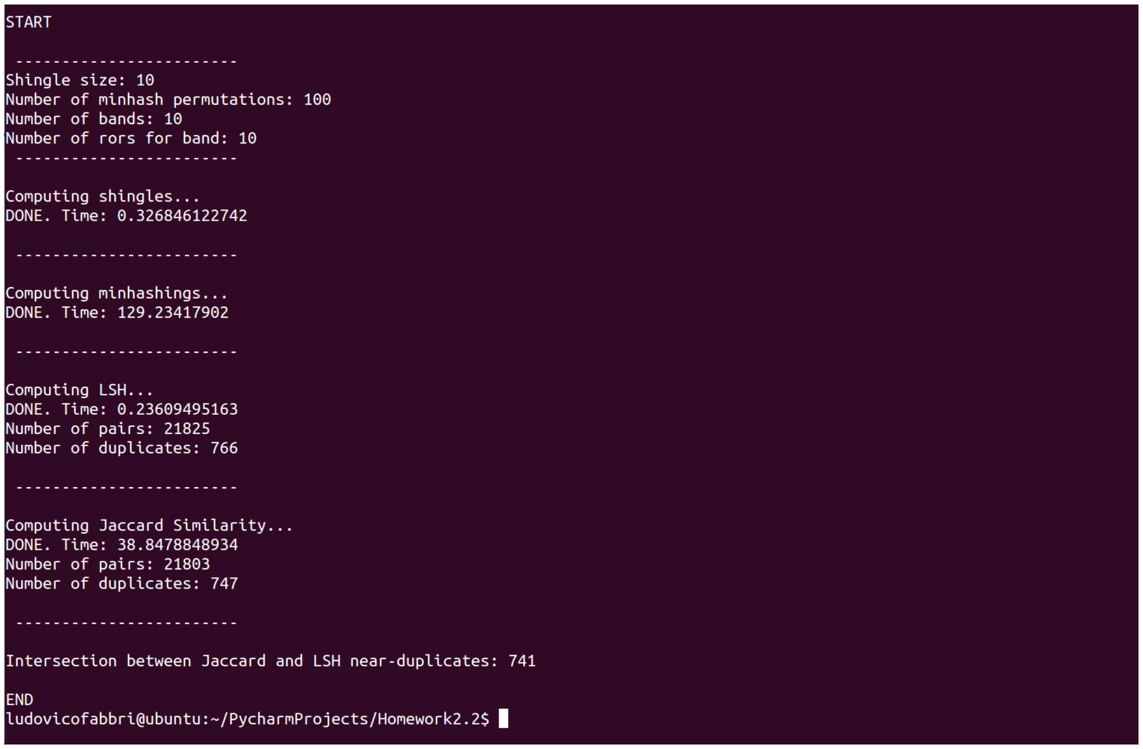
Real solution:

$x \approx 9.78385$, $y \approx 10.2209$

Figure 1: Wolfram computing b and r in LSH with n=100 and t=0.8

So i've choose b=10 and r=10.

And this is the output of the whole program:



```
START

 -----------------------
Shingle size: 10
Number of minhash permutations: 100
Number of bands: 10
Number of rors for band: 10
 -----------------------

Computing shingles...
DONE. Time: 0.326846122742

 -----------------------

Computing minhashings...
DONE. Time: 129.23417902

 -----------------------

Computing LSH...
DONE. Time: 0.23609495163
Number of pairs: 21825
Number of duplicates: 766

 -----------------------

Computing Jaccard Similarity...
DONE. Time: 38.8478848934
Number of pairs: 21803
Number of duplicates: 747

 -----------------------

Intersection between Jaccard and LSH near-duplicates: 741

END
ludovicofabbri@ubuntu:~/PycharmProjects/Homework2.2$
```

Figure 2: output for n=100, b=10, r=10, t=0.794

- Time of LSH: 0.236
- Number of near-duplicates found with LSH: 766
- Time of Jaccard similarity: 38.847
- Number of near-duplicates found with JS: 747
- Size of the intersection: 741

2 Problem 2

2.1

Recall that the k-means cost function for clustering C is the:

$$J = \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - \mu_k\|_2^2 \tag{1}$$

where $C = \{C_1, C_2, ..., C_k\}$ is a clustering of the dataset V.

We want to minimize the l_1 distance between points of the dataset and the relative cluster center in this objective function:

$$J = \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - \mu_k\|_1 \tag{2}$$

where K is the set of clusters, x_i are the points of dataset and μ_k are the centroids of clusters. We can re-write this function in another way:

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \cdot \|x_i - \mu_k\|_1 \tag{3}$$

Where N is the set of points of the dataset and r_{nk} is a binary indicator variable defined as follow:

$$r_{nk} = \begin{cases} 1 & \text{if the n-th point is assigned to the k-th cluster} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Our goal is to find values for $\{r_{nk}\}$ and $\{\mu_k\}$ that minimize J.
To do this we can use an iterative approach in which each iteration is divided in two successive steps:

1. First we optimize x_n keeping μ_k fixed
2. Than we optimize μ_k keeping x_n fixed

We can repeat this procedure until convergence for some threshold parameter. In the first step we have centroids fixed and want to minimize J as function of the distribution of the point x_n, that is find r_{nk}. Since each point of the dataset is independent from the others, we can choose r_{nk} to minimize the manhattan distance (norm-1 distance or l_1) between the point x_n and the centroid μ_k. So we have that:

$$r_{nk} = \begin{cases} 1 & \text{if k} = \text{arg } min_j \left\| x_n - \mu_j \right\|_1 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Because we are in an ordinated space (otherwise manhattan distance has no sense), assuming that $\rho(x_1, \mu_1)$ is the manhattan distance between the point x_1 and the centroid μ_1 and $\rho(x_1, \mu_2)$ is the distance between the same point and the centroid centroid μ_2, we can say that if $\rho(x_1, \mu_1) < \rho(x_1, \mu_2)$, than also the euclidean distance (2-norm or l_2) between point x_1 and the centroid μ_1 is lower than the euclidean distance between x_1 and the centroid μ_2. That is, we can re-write r_{nk} using the l_2 euclidean distance:

$$r_{nk} = \begin{cases} 1 & \text{if k} = \text{arg } min_j \left\| x_n - \mu_j \right\|_2 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

So far, we found that the first step of the optimization algorithm will find the same r_{nk} using the objective function of k-means in the (1) which uses the 2-norm distance or the objective function in the (2) which uses the 1-norm distance.

Now let's focus on the second step. We want to find values for centroids μ_k that minimize J keeping r_{nk} fixed (so the data points are assigned). So we want to resolve this problem:

$$min(\sum_{n=1}^{N} r_{nk} \cdot \|x_n - \mu_k\|_1) \qquad \forall k \in K \tag{7}$$

To find the minimum we can calculate the derivative of the function above with respect to μ_k and set it equal to zero. We know that the derivative of the absolute value is the sign function (excluding the point where argument is zero). That is:

$$\frac{\partial J}{\partial \mu_k} = \frac{\partial \sum_{n=1}^{N} r_{nk} \cdot \|x_n - \mu_k\|_1}{\partial \mu_k} = 0 \tag{8}$$

$$\frac{\partial J}{\partial \mu_k} = \sum_{n=1}^{N} r_{nk} \cdot sign(x_n - \mu_k) = 0 \qquad \forall k \in K, \mu_k \neq x_n \tag{9}$$

So for each cluster we want to find μ_k that makes a partition for that cluster such that we will have the sum of the sign functions equal to zero. Among the infinite solutions we can find, we want to find the one that minimize the absolute error between the data-points of the cluster. We can recall a well known property of the median, which is to minimize the sum of the mean absolute errors of a set of values x_i from another generic value c:

$$\sum_{i=1}^{N} |x_i - M_e| \leq \sum_{i=1}^{N} |x_i - c| \tag{10}$$

From this we can easily see that the solution for the (7) is that μ_k must be the median of the points assigned to the k-th cluster:

$$\mu_k = \text{Median}\{x_i\} \qquad \forall r_{ik} = 1, x_i \in V, k \in K \tag{11}$$

So here we have found a substantial difference with the k-means algorithm which for each cluster find the centroid as the mean of the points assigned to that cluster.

2.2

Assume that the optimal solution for the k-means cost function has cost C. You are now asked to cluster the points, but under the constraint that the cluster representative (i.e., the point corresponding to μ_k) has to be one of the input points in V . Prove that there exists a solution with cost at most 4C.

We can write the cost function of k-means as follows:

$$C = \sum_{n=1}^{N} \|x_n - \mu_k\|^2 \qquad \forall k \in K \tag{12}$$

where μ_k is the centroid of the k-th cluster and doesn't have to be one of the x_i (a point of the dataset).
Now let's define a cost function where instead of μ_k we pick a point of the dataset as the centroid, let's call it x^c_{nk}:

$$C' = \sum_{n=1}^{N} \|x_n - x_{nk}^c\|^2 \qquad \forall k \in K,\ \ x_{nk}^c \in V \tag{13}$$

From the triangular inequality we can say that if x,y,z are the points of a triangle than is true that $\rho(x,y) \leq (\rho(x,z) + \rho(y,z))$, where $\rho(a,b)$ is the distance between point a and point b.

Also we can write that:

$$(x+y)^2 \leq 2(x^2+y^2) \tag{14}$$

In fact:

$$\begin{aligned} x^2+y^2+2xy &\leq 2x^2+2y^2 \\ x^2+y^2-2xy &\geq 0 \\ (x-y)^2 &\geq 0 \end{aligned} \tag{15}$$

which is always true.

Now we can consider a triangle formed by the following points respectively: x_n, x_{nk}^c and μ_k, where x_n is a generic point of the dataset, x_{nk}^c is the point of the dataset assigned to the k-th cluster and μ_k is the centroid found by the k-means algorithm for the k-th cluster (that is the mean of the points of the k-th cluster).

That said, we can now easily write the following inequality:

$$C' = \sum_{n=1}^{N} \|x_n - x_{nk}^c\|^2 \leq 2\sum_{n=1}^{N}(\|x_n - \mu_k\|^2 + \|\mu_k - x_{nk}^c\|^2) \leq$$
$$\leq 2\sum_{n=1}^{N} \left\|x_n - \mu_k)^2\right\| + 2\sum_{n=1}^{N} \|\mu_k - x_{nk}^c\|^2 \tag{16}$$
$$\forall k \in K, \;\; x_{nk}^c \in V$$

From the last two terms we can observe that:

$$2\sum_{n=1}^{N} \|x_n - \mu_k\|^2 = 2C$$
$$2\sum_{n=1}^{N} \|\mu_k - x_{nk}^c\|^2 \leq 2C \tag{17}$$
$$\forall k \in K, \;\; x_{nk}^c \in V$$

because x_{nk}^c is a generic point of the dataset that we have choose as the centroid of the cluster k.

Thus we finally found that:

$$\sum_{n=1}^{N} \|x_n - x_{nk}^c\|^2 \leq 2C + 2C \leq 4C \qquad \forall k \in K, \;\; x_{nk}^c \in V \tag{18}$$

which means that:

$$C' \leq 4C \tag{19}$$