Big Data Analytics Programming
Assignment 2: *Locality sensitive hashing*

Toon Van Craenendonck
toon.vancraenendonck@cs.kuleuven.be

Jessa Bekker
jessa.bekker@cs.kuleuven.be

Collaboration policy:
Projects are independent: no working together! You must come up with how to solve the problem independently. Do not discuss specifics of how you structure your solution, etc. You cannot share solution ideas, pseudocode, code, reports, etc. You cannot use code that is available online. You cannot look up answers to the problems online. If you are unsure about the policy, ask the professor in charge or the TAs.

1 Problem statement and motivation

Finding sets of similar objects in large collections is a simple problem with a variety of applications. Examples include identifying people with similar movie tastes, identifying similar documents to avoid plagiarism, ... The problem can be specified as follows: given a similarity function sim that maps every two objects to a numeric score, find all the pairs of objects (x, y) such that $sim(x, y) > t$, where t is a user-defined similarity threshold.

The problem can be solved with the following (naive) algorithm.

Find similar pairs:
Input:
A collection $C = \{x_1, \ldots, x_n\}$ of objects
A similarity function $sim : C \times C \rightarrow \mathbb{R}$
A minimum similarity threshold t
Output:
The set S of all the pairs (x_i, x_j) of similar objects. (i.e. pairs such that $sim(x_i, x_j) > t$)

for all $i \in 1 \ldots n$ **do**
  for all $j \in i + 1 \ldots n$ **do**
    if $sim(x_i, x_j) > t$ **then**
      $S = S \cup \{(x_i, x_j)\}$
    end if
  end for
end for

Observe that the complexity of this algorithm is quadratic with the number of objects in the collection. Such a complexity is acceptable for simple problems, but quickly becomes unpractical as the number of objects grows. For large collections of objects, such as collections of webpages or movie recommendations with several million entries, computing the similarities using this algorithm can take weeks.

Locality sensitive hashing (LSH) is an approximate method that can be used to find pairs of objects with

high similarity more efficiently. It hashes the most similar objects into the same buckets, and most dissimilar objects into different buckets. Approximate similarity can then be computed for only those pairs of objects that are hashed into the same bucket. If the hashing function is properly designed, this can yield a drastic reduction of complexity.

More details, and a complete description of the locality senstive hashing procedure can be found in the book *Mining of Massive Datasets* by Anand Rajaraman and Jeffrey David Ullman. The book is freely available online and the corresponding chapter about locality sensitive hashing can be found at:

`http://infolab.stanford.edu/~ullman/mmds/ch3.pdf`

Read this chapter carefully before you continue reading the following sections.

2 Applying LSH to twitter data

For this assignment you will implement LSH and apply it to a dataset containing tweets.

2.1 The data

We will consider 7416113 tweets, obtained from `https://archive.org/details/twitter_cikm_2010`. The tweets are available in `/cw/bdap/assignment2/tweets`. Each line in the file has the following format: `UserID \t TweetID \t Tweet\t CreatedAt`. We are only interested in the third column, containing the actual text.

2.2 Given code

You are given code that is able to read these tweets, and obtain similar pairs using the naive brute force algorithm described above. More specifically, you are given the following files (in `assignment2.tgz` on Toledo):

- `Shingler.java`: maps string documents to a shingle representation
- `BruteForceSearch.java`: implementation of the naive algorithm for finding similar documents
- `Runner.java`: allows to run the brute force search from the commandline
- `SimilarPair.java`: convenience class to represent similar pairs
- `TwitterReader.java`: used by BruteForceSearch to read tweets
- `Primes.java`: used to obtain the least prime number greater than a certain value, needed for constructing a hash table
- `MurmurHash.java`: hash function used in the Shingler, might also be used in the LSH implementation

You can run the code with the following command:

```
java Runner -threshold 0.5 -maxFiles 100 -inputPath path/to/tweets
                           -outputPath path/to/output -shingleLength 3 -nShingles 1000
```

Running this command produces an output file containing the pairs with a similarity above the given threshold. Each line has the following format: `tweetID1 \t tweetID2 \t similarity`. It is important to note here that the identifiers for the tweet correspond to the line numbers in the `tweets` file, not to the tweet identifiers used in that file.

2.3 Your task

Your task is to implement the LSH algorithm, and run it on the twitter data to find similar pairs of tweets. You are free to use the code given in `assignment2.tgz`, but you don't have to (except for `Shingler.java` for running on the full dataset, as discussed in the next section). Given your implementation, you should deliver (1) an analysis of the effect of the most important parameters, (2) a commandline command corresponding to a specific parameter setting to run on the full dataset. These two deliverables are discussed in more detail below.

2.3.1 Parameter analysis

Identify the most important parameters in your program, and provide an analysis of the effect of these parameters on both runtime and solution quality. With the latter, we mean the number of true positives and false positives that your LSH program produces. This analysis should be based on a set of experiments performed on a subset of the twitter dataset (as you need to be able to run the naive algorithm to get the correct list of similar pairs).

2.3.2 Running on the full dataset

Find a good parameter configuration for obtaining the pairs of tweets in the full dataset that have a **Jaccard similarity > 0.9** when using **3-shingles** at the character level. You should provide this parameter configuration through a command given in a README file contained in your solution, i.e. we expect a command of the following form:

```
java MyLSHRunner -threshold 0.9 -maxFiles 7416113 -shingleLength 3
                    -inputPath path/to/tweets -outputPath path/to/output
```

Besides the arguments that are specified in the command above, you should also expose all other parameter that are important to the runtime or solution quality of your program (for example, we would certainly expect to see parameters specifying the number of bands and the number of rows per band). Discuss the choice of these parameters in your report.

Executing the above command should result in an output file `path/to/output`, in which each line has the following format: `tweetID1 \t tweetID2 \t similarity`. The output file should contain all the pairs of items that are identified as having a similarity > 0.9 by your LSH program (this file can contain false positives). The similarity listed in the third column does not have to be exact (i.e. it can be computed based on the signature matrix).

Some important remarks:

- For this part of the assignment you should use the provided `Shingler` implementation.
- For this part of the assignment you should not use any additional data preprocessing (as we will check your solution for correctness, which requires using the exact same input data).
- Your solution should compile and run on the departmental machines.
- Your solution should require $<$ 2GB of memory. Note that this is the memory limit for remote (ssh) users of the departmental machines. This is an important limitation, take it into account while developing your solution!
- In your output file tweets should be identified by their line numbers in the `tweets` file. I.e. `tweetID1` and `tweetID2` in the above format refer to line numbers containing the tweets (starting from line 0), not to the tweet identifiers listed in the input file.

- This part will be graded on both correctness (in terms of number of true positives and false positives produced by your program), and runtime efficiency.

3 Report

Your report should contain (1) a discussion of any important (w.r.t. runtime and/or memory efficiency) choices that you had to make in developing your solution of at most 1 page, (2) a thorough description of the experiments performed as discussed in 2.3.1, of at most 1.5 pages excluding any plots or tables, and (3) a brief motivation for the specific set of parameters chosen to run on the full dataset, as well as the runtime of your final solution.

4 Important remarks

- The assignment should be handed in on Toledo before February 16th. There will be a 10% penalty per day, starting from the due day.
- Do not upload any data files or result files.
- You must upload an archive (`.tar.gz`) containing the report as a PDF file, a README text file containing at least the command to run on the full dataset, and a folder with all the source files (see hierarchy below).

```
assignment2/
├── report.pdf
├── README
└── src/
    ├── Shingler.java
    ├── TwitterReader.java
    └── ...(all source files needed to compile and run your solution)
```

- The point distribution is: 24 points on implementation, 6 on report

Good luck!