Big Data Analytics Programming
Assignment 3: *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 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 $sim(x, y) \geq t$, 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 $sim(x_i, x_j) \geq t$)

```
for all i ∈ 1...n do
    for all j ∈ i+1...n do
        if sim(x_i, x_j) ≥ t then
            S ← S ∪ {(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. Once all the objects have been sorted into buckets, one simply has to compute parwise similarities among the pairs of objects that have been sorted in 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 Application 1: document similarity detection (7 pts)

Your goal is to implement and tune the locality sensitive hashing algorithm, and evaluate its performance to identify the most similar documents in a large collection of documents.

2.1 LSH implementation

The similarity measure that we will use to compare documents is the *Jaccard* distance. It is defined as follows for two objects represented as sets, X and Y respectively.

$$sim(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} \tag{1}$$

Shingles: Text documents are typically sequences of words (a.k.a sentences) rather than sets. In order to be able to apply the *Jaccard* distance measure, one must first convert the documents into sets of *shingles*. A shingle is a short fraction of the document that can appear in other documents. Once a set of shingles has been defined for the collection of documents, each document can be represented as the set of shingles it contains. Obviously, the length of the shingles and the procedure used to break sentences down into shingles will have an important impact on the final quality of the results. We provide a basic shingle definition (See `SimpleShingler.java`).

Minhash signature matrix: The first step to build the signature matrix is to define n hash functions $h_1, \ldots, h_n$. Each hash function h_j will represent an (approximate) permutation of the row indexes in the characteristic matrix required to compute one minhash. Practically, this requires writing a procedure that, given the number of rows r in the characteristic matrix and the desired number of hash functions n, will produce a 2D array `H` of dimension $r \times n$ such that `H[i][j]` $= h_j(i)$. Here $h_j(i)$ is the result of the hashing function h_j applied to the the row index i. You need to implement this procedure in the `constructHashTable(...)` method of the `LSH.java` class.

The characteristic matrix and the hash values can now be used to compute the minhash *signature matrix* (as described in the book chapter). The signature matrix will be represented as a 2D array `S` of dimension $n \times c$ where each element `S[i][j]` is the minhash signature i of the document j (c is the number of documents.) This procedure must be implemented in the `constructSignatureMatrix` method of the `LSH.java` file.

Sorting documents in buckets: At the core of the LSH algorithm is the `lsh(...)` method, which is used to sort documents with similar signatures into the same buckets. The method breaks the signature matrix into m bands and r rows (as described in the referred book chapter) and computes, for each band, a new hash value for all the document signatures. (We use a `String` to represent the hash value.)
Be careful, there is one hash map per *band*, that is why the `lhs(...)` method returns an object of type `List<Map<String, Set<Integer>>>`.

Testing candidate pairs of similar documents: The final step is to compute the *Jaccard* similarity of all the candidate pairs of documents and store them in the result set of type `Set<SimilarPair>` only if their similarity is above the specified threshold. This must be implemented in the `getSimilarPairsAboveThreshold` method. Note that every pair of documents that appear in the same bucket and that is not similar enough is a false positive.

You can download the code for this assignment on Toledo (`assignment3.tgz`).

List of files in `assignment3/src`:

- The `DocumentHandler.java` class (complete). Can be used to transform input documents into sets of shingles.
- The `SimpleShingler.java` class (complete) is a basic class to break down documents into shingles.
- The `BruteForceSearch.java` (complete) implementation of the naive algorithm: simply compares all the pairs individually. It can be used to test your implementation on simple datasets, or as a baseline.
- The `DocumentRunner.java` (complete) parses the parameters and runs the similarity searcher.
- The `LSH.java` (incomplete!) which is a stub for your implementation of LSH. In this class, you must implement the following functions as described above:
  - `constructHashTables(...)`
  - `constructSignatureMatrix(...)`
  - `lsh(...)`
  - `getSimilarPairsAboveThreshold(...)`
  - `getNeighborsAboveThreshold(...)` (Required for Section 3 only.)
- ...

Please, **do not** modify the signature of these functions and use the output variables as expected (e.g. do not permute the dimensions in the output arrays). You are free to modify the rest of the code as long as these methods and the executable file (see below) work as expected.

2.2 Running and testing

You can test the naive approach (which is provided) and your implementation of the LSH algorithm using the Reuters text data available online[1].

[1] `https://people.cs.kuleuven.be/~toon.vancraenendonck/bdap/reuters.tgz`

For example, if you extract the `reuters.tgz` archive in the `data/` directory, you can run the naive algorithm on the first 100 files with the following command.

```
java DocumentRunner -threshold 0.5 -dir ../data/reuters -maxFiles 100 -method bf -shingleLength
10
```

Where:

- `-dir` *path*: is the data directory.
- `-maxFiles` *int*: is the maximum number of files to consider in *path*. If set to n, `DocumentHandler.java` will only load documents $0, \ldots, n-1$.
- `-treshold` *double*: similarity threshold.
- `-method bf` *or* `lsh`: method used. Either `bf` for *brute force* or `lhs` for *locality sensitive hashing*. In the original version of the archive, only the `bf` method is implemented.
- `-shingleLength` *int*: length of the shingles.

If the method is `lsh`, the following arguments are also mandatory.

- `-numHashes` *int*: the number of hash functions for the signature matrix.
- `-numBands` *int*: the number of bands.

You are free to add new parameters if necessary.

2.3 Tuning and evaluation

To evaluate the quality of your algorithm, measure the number of false positives (i.e. number of pairs tested for *Jaccard* that were below the threshold) and the number of true negatives (i.e. number of pairs not tested whose *Jaccard* distance is above the similarity threshold). Tune the parameters for this dataset and report your experiments. Write a `README` file with the exact command that yielded good results with these tuned parameters (after compiling your code we should be able to simply execute this command).

3 Application 2: movie recommendation (7 pts)

This part of the assignment is based on the same LSH algorithm as the previous one. Instead of similar documents, we now search for similar users and use these to make movie recommendations.

3.1 Task description

Movie Lens is a collection of large movie recommendation datasets. Your goal is to reuse the LSH class defined in the previous section to make a simple movie recommender. The training set that we will use contains 10^6 ratings by $69,878$ users over $10,667$ movies and is available online[2]. Download and extract the archive in the `data/` directory. More datasets and information about Movie Lens can be found at `http://grouplens.org/datasets/movielens/`.

[2] `https://people.cs.kuleuven.be/~toon.vancraenendonck/bdap/movielens.tgz`

List of files in `assignment3/src/` (cont.): The following extra files are provided for this part of the assignment.

- `MovieHandler.java` class (complete). Provides a set of utility functions to access the rating data, and converts the ratings to a set representation for every user.
- `MovieRunner.java` (incomplete!). Main file to run and evaluate the movie predictor. In this file you must implement the following method.
  - `predictRating(userID, movieID)`. Estimate and return the rating of `userID` for movie `movieID`, you must complete this method.

3.2 Running and testing

The following command should train your recommender on the `ra.train` data and evaluate its performance on the `ra.test` testing data.

```
java MovieRunner -trainingFile ../data/movielens/ra.train -testFile ../data/movielens/ra.test
```

You can already execute the recommender, but the predictions will not be good as the `predictRating` method always returns the same prediction (2.5/5).

You may add any optional arguments. Please include the command with the optimal parameter settings in the `README` file (we should be able to simply run this command after compiling your code).

3.3 Parameter tuning and evaluation

At this stage, your algorithm should have a variety of parameters (number of hashes, number of bands, hashing functions, aggregating method, default prediction, etc.). Understanding the impact of each parameter and finding the optimal combination of parameters for the problem at hand is an important, and difficult aspect of every data analysis process. Run experiments and find the best combination of parameters for this application. Do not forget to discuss these experiments in the report. Note that your system will be evaluated using the parameters that you provide in your `README` file.

In order to evaluate the quality of your predictions, we will use Root Mean Squared Error (RMSE):

$$RMSE = \sqrt{\frac{\sum_i^n (\hat{y}_i - y_i)^2}{n}}$$

Where n is the number of testing examples, $\hat{y}_i$ is the predicted rating for the movie i and y_i is the actual rating for this movie. The $RMSE$ of your recommender should be compared to the baseline: predicting the average rating for this movie in the dataset.

Beware! Predicting movie ratings is inherently difficult because the linking of a movie involves a variety of human factors that are both complex and not reflected in the database. Unlike for more simple tasks, advanced techniques only improve the accuracy of the prediction by small margins. As a matter of fact, the Netflix competition awarded USD 1,000,000 to improve the baseline by only 20% and the recommender system that won the challenge was the result of several years of research and tuning.

For reference, a very basic solution for this exercise outperforms the baseline by $\sim 4\%$ on RMSE.

4 Report (6 pts)

Write a report to describe your implementation and to present the experiments that you ran to tune parameters and to evaluate the performance of your algorithm(s). The text of the report (minus any figures and tables) should not exceed four pages. You are free (and encouraged) to include any figures and tables that you find interesting and instructive. The report should accomplish two things. First, it should highlight and discuss any important design decisions you have made in your implementation. Second, it should describe the experimental methodology, and then describe and analyze the results. The analysis part is important.

Additionally, if you suspect that your project has a bug, the report is a good place to discuss it. You can state why you think there is a bug, why it may have arisen, what you've tried to find fix it. This allows us to help with grading.

5 Important remarks

- The assignment should be handed in on Toledo before April 11th. As usual, there will be a 10% penalty per day, starting from the due day.
- **Do not** upload any data file (i.e. the data directory must remain empty).
- Do not forget the `README` file, with one command line for the document similarity checker and one for the movie recommender. These will be used for evaluation.
- You must upload an archive (`.zip` or `.tar.gz`) containing the report (as pdf) and a folder with all the source files (See file hierarchy below.) Feel free to add as many files or directories as you need. However, if you think this extra content is valuable and deserved to be looked at, please describe why in the report.

```
assignment3/
├── report.pdf
├── README
├── src/
│   ├── SimilaritySearcher.java
│   ├── BruteForceSearch.java
│   ├── LSH.java
│   ├── DocumentHandler.java
│   ├── SimilaritySearcher.java
│   ├── DocumentRunner.java
│   ├── MovieHandler.java
│   ├── MovieRunner.java
│   └── ...
└── data/
    └── <emtpy>
```

Good luck!