TI2736-C Datamining
Assignment 1: Finding Similar Items

Delft University of Technology, February–April 2015

Thomas Abeel, Marcel Reinders
Zekeriya Erkin, Julian Kooij
Daan Rennings, Tom Viering

Pattern Recognition and Bioinformatics Group

based on work from

Laurens van der Maaten, Hans Gaiser, Daan Rennings

TU Delft

1 Finding Similar Items

In these exercises, you will create algorithms in Java for finding similar items in a dataset. The template source code required for these exercises can be found on Blackboard. **Please note that you have to be able to answer the questions stated throughout the exercises, though it is not needed to hand in your answers to these questions separately.** Also, if you also followed this course last year, please note that there have been some minor changes to the exercises.

Exercise 1.1. Shingles

The `ShingleSet.java` file contains a useful template that can be used to complete this exercise. The `ShingleSet` class extends a Java TreeSet and is intended to contain shingles from some text, but some parts of the code are missing.

Step 1. First we will implement the method `shingleString`. This method takes as argument some string and cuts it up in shingles of size `k`.

For example, if the input string is:

"`abcdabd`"

The resulting `ShingleSet`, with a `k` of 2 will be:

{"`ab`", "`bc`", "`cd`", "`da`", "`bd`"}

Implement this method and verify that it works as intended.

Question 1.1. What happens when you try to add shingles that are already in the `ShingleSet`? In the above example, the shingle "`ab`" occurs twice in the string, how many times does it occur in the `ShingleSet`?

Step 2. Next we will be implementing the `jaccardDistance` method in the `ShingleSet` class. This method takes as input argument some other `TreeSet` and computes the Jaccard distance between this set and the given set. Remember that the Jaccard distance can be calculated as follows:

$$d(A, B) = 1 - \frac{| A \cap B |}{| A \cup B |} \tag{1}$$

Hint: Use the method `retainAll` for computing the intersection of two `ShingleSets` and `addAll` for computing the concatenation of two `ShingleSets`.

Step 3. Create two separate `ShingleSets` called `s1` and `s2` in the `exercise1_1()` method in `main.java`, with `k` set to 5. Add the following string using the method created in step 1 for set `s1`:

"`The plane was ready for touch down`"

Do the same for `s2`, but with the following string:

"The quarterback scored a touchdown"

Use the method created in step 2 to calculate the Jaccard distance between these two sets and verify your results.

Question 3.1. Are these sentences very similar? Should the Jaccard distance between these two sets therefore now be large or small?

Question 3.2. We had previously set our `k` to 5, what would happen if we reduce our `k` to 1? Would that increase or decrease the distance between our two sets? Why is that? What happens in the case where we increase our `k` to 15? Theorize what a feasible value for `k` would be, given the sentences above.

Step 5. Both sentences contain whitespaces, but these would not appear to contribute much to the actual meaning of the sentence. An option would be to strip all whitespaces from the sentences before cutting them into shingles. Copy the contents from the method of step 1 to the method `shingleStrippedString`. Before creating any shingles in this method, remove all whitespaces from the string.

Step 6. Create two new `ShingleSets` called `s3` and `s4` and fill them in a similar manner as you did for `s1` and `s2`, but now use the method from step 5.

Question 6.1. Did the Jaccard distance between the two sets now increase or decrease? Why is that?

Exercise 1.2. Minhashing

For this exercise you need to modify the given `MinHash.java` file. At the end of this part of the exercise you can create a minhashing signature matrix of `ShingleSets`.

Step 1. In `main.java`, create 4 `ShingleSets`, `s1-s4`, with `k` set to 1. The shingles in these sets are as follows:

```
s1 = {"a","d"}, s2 = {"c"}, s3 = {"b","d","e"}, s4 = {"a","c","d"}
```

Step 2. Create a `MinHash` object in the `main` method and add two hash functions to this object using the `addHashFunction` method. The functions should hash in the following way:

$$h_1(x) = x + 1 \mod n \quad (2)$$
$$h_2(x) = 3x + 1 \mod n \quad (3)$$

Where x is the input variable and n is the number of unique shingles of all sets (which does not need to be set right away).

Hint: For $x = 3$ and $n = 2$, $h_1(x) = x + 1 \mod n$ would result in 0, which would be the hashed value of x.

Step 3. Next we are going to create the method `MinHash.computeSignature` that will create the minhash signature matrix from our sets `s1-s4` using our hash functions h_1 and h_2. In `MinHash.java`, complete the code for the `computeSignature` method. You could make use

of the pseudocode below.

foreach *shingle x in the shingle space* **do**
  foreach *ShingleSet S* **do**
    if $x \in S$ **then**
      foreach *hash function h* **do**
        signature(h, S) = $\min(h(x), \text{signature}(h, S))$
      end
    end
  end
end

Algorithm 1: Pseudocode for the `computeSignature` method.

Step 4. Add the previously created `ShingleSets` to the `MinHash` object.

Question 4.1. Verify that the result of your implementation is correct, given the hashing functions and `ShingleSets` above.

Exercise 1.3. Locality Sensitive Hashing

In this part of the exercise we will use the `ShingleSets` and `MinHash` classes to compute a Locality-Sensitive Hashing table using the banding technique for minhashes described in the lecture and in the book. For this you will need to modify the `LSH.java` file.

Step 1. In `main.java`, remove the hashing functions used in the previous exercise and use the `MinHash.addRandomHashFunctions` method to add 100 random hashing functions.

Step 2. In `LSH.java`, complete the missing code in the `computeCandidates` method. For this you may use the pseudocode given below. Also note that the Java `Object` has a `hashCode()` method, which hashes a given object.

Hint: You may want to use the `MinHashSignature.colSegment` method.

```
// store all items in LSH
initialize buckets as a list of lists of integers
foreach band do
  foreach set do
    extract a column segment of length r, for this band and set, as string s
    add s to buckets[hash(s)]
  end
end

// retrieve candidates from LSH
foreach band do
  foreach set do
    extract a column segment of length r, for this band and set, as string s
    retrieve all items in buckets[hash(s)] and add these items to the list of candidates for
    this set
  end
end
```

Algorithm 2: Pseudocode for the `computeCandidates` method.

Step 3. Similarly as before, compute the minhash signature matrix using the 100 random hash

functions. Use this result to compute candidate pairs that might have high similarity. Use a bucket size of 1000 and 5 rows per band.

Question 3.1. What happens if the number of buckets is too small? For example what would happen if we only use 10 buckets?

Question 3.2. What is the effect of the number of rows per band? If we set the number of rows per band to 1, what will happen? And if you set the number of rows per band to the length of the signature?

Step 4. Iterate through the set of candidate pairs and for each pair, check whether their Jaccard distance is smaller than some threshold `t` (for example 0.5). If this is the case, output that pair of indices to the console. Run the code a number of times and verify that your result is correct.