TI2736-C Datamining
Assignment 1: Finding Similar Items

Delft University of Technology, February–April 2017

Thomas Abeel, Marcel Reinders
Zekeriya Erkin
Wouter Kouw, Tamim Abdelaal
Lucas van Dijk, Samuel Sital, Ravi Autar

Pattern Recognition and Bioinformatics Group

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.** While it is not needed to hand in your answers to these questions separately, **do write the answers down** so that you have them ready when the lab assistant checks your work.

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 4. 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 5. 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 4.

Question 5.1. 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). For example: for $x = 3$ and $n = 2$, $h_1(x) = x + 1 \mod n = 0$, which is the hashed value of x.

Question 2.1. To gain some insight in computing minhash signature matrices, compute the MinHash signature by hand of the given `ShingleSets` above using the the hash functions h_1 and h_2. Do this computation by hand! Refer to the slides or study material if you forgot how to do this.

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) = $\min(h(x),$ 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. Compute the minhash signature matrix using your program. Verify that the result of your implementation is correct by comparing the answers of your program to the answers that you got in Question 2.1.

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 lists of integers
foreach band do
    foreach set do
        s = a column segment of length r, for this band and set
        add set to buckets[hash(s),band]
    end
end

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

Algorithm 2: Pseudocode for the `computeCandidates` method.

Note: the list of candidates thus should be pairs of *two* (different) numbers. Check that you cannot get candidates that are pairs of three sets (or more), or candidates that consists of just a single set. Also double check that you do not have duplicate candidates.

Question 2.1. An important implementation issue is that you should keep seperate lists of buckets for each band. This means that this algorithm will work suboptimal if you index the buckets only as: `buckets[hash(s)]` instead of `buckets[hash(s),band]`. Why is this the case?

Step 3. Similarly as before, compute the minhash signature matrix using the 100 random hash functions. Use a bucket size of 1000 and 5 rows per band.

Question 3.1. When you run your code multiple times, you will notice that sometimes you get other candidates. How is this possible?

Question 3.2. Run your code 10 times. Take notes which candidates get suggested and how many times each candidate gets suggested. How does this relate to the Jaccard distance between the two sets of the candidate pair (not in terms of formulas, just an indication)? To check this, compute the Jaccard distance between all possible combinations of all ShingleSets and compare this to the frequencies (how many times a pair gets suggested as a candidate) to verify your idea.

Question 3.3. Why (or when) would you use this algorithm?

Question 3.4. What happens if the number of buckets is too small? For example what would happen if we only use 10 buckets?

Question 3.5. 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?