Intelligent Data Management - Exercise 1

Christoph Prinz

October 19, 2017

Assignment 1

Exercise a

Term i appears in n_i of the N documents

Appears in 40 documents: $IDF_i = log_2(\frac{N}{n_i}) = log_2(\frac{10000000}{40}) = 18$

Appears in 10000 documents: $IDF_i = log_2(\frac{N}{n_i}) = log_2(\frac{10000000}{10000}) = 10$

Exercise b

Given the occurrence of a term i in document j is f_{ji} and $max_k f_{kj}$ the maximum number of occurrences of any term in this document, the term frequency TF is defined as:

$TF_{ij} = \frac{f_{ij}}{max_k f_{kj}}$

Word w appears in 320 documents. In document d the maximum occurence of any word is 15.

a) w appears once:

$TF_{wd} = \frac{1}{15}$

$IDF_w = *log_2 \frac{10000000}{320} = 15$

$TF.IDF = \frac{1}{15} * 15 = 1$

b) w appears five times:

$TF_{wd} = \frac{5}{15} = \frac{1}{3}$

$IDF_w = *log_2 \frac{10000000}{320} = 15$

$TF.IDF = \frac{1}{3} * 15 = 5$

Exercise c

The basic rule is, that the possible hash-keys should not have any common factor with B (15 in this example). The population should therefore not be generated with a c of 3 or 5. To achieve an equal distribution of the generated hashs, c should be set to 1.

Assignment 2

Exercise a

The implementation consists out of a class `Document`, that counts words in a document and calculates the term-frequency of a given term in this document. The class `DocumentCollection` reads multiple input files and calculates IDF and TF.IDF for a given document and a given term. The source code is printed on the following pages.

Exercise b

The choice of parameters is explained with comments in the source code.

```java
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

public class Document {
    private Map<String, Integer> wordCount = new HashMap<String, Integer>();

    public Document(String path) throws IOException {
        countWordsFromFile(path, Charset.forName("Cp1252"));
    }

    public boolean doesTermAccur(String term){
        return wordCount.containsKey(term);
    }

    /**
     * calculate term frequency for a given term in document
     * @param term: term w to calculate tf
     * @return value for tf
     */
    public double get_tf(String term){
        int f_ji = 0;
        if (wordCount.containsKey(term)) {
           f_ji = wordCount.get(term);
        }
        int max_k_fkj = Collections.max(wordCount.values());

        return calculate_TF((double)f_ji, (double) max_k_fkj);
    }

    /**
     * calculates tf
     * @param f_ji: number of occurrences of a term
     * @param max_k_fkj: maximum number of occurrences of any term
     * @return value for tf
     */
    private static double calculate_TF(double f_ji, double max_k_fkj){
        return f_ji/max_k_fkj;
    }

    /**
     * Method to count words from a text file
     * @param path: path of file to read
     * @param encoding: encoding of file
     * @throws IOException if file can't be loaded
     */
    private void countWordsFromFile(String path, Charset encoding)
            throws IOException {
        String text = new String(Files.readAllBytes(Paths.get(path)), encoding);
        //Trim file in order to get only raw words
        text = text.replaceAll("[\\s]", " ");
        text = text.replaceAll("[^A-zäöüÄÖÜß ]", "");
        text = text.toLowerCase();
        String[] parts = text.split(" ");

        //count words
        for (String part : parts) {
            if (wordCount.containsKey(part)) {
                wordCount.put(part, wordCount.get(part) + 1);
            } else {
                wordCount.put(part, 1);
            }
        }
    }
}
```

```java
import java.util.HashMap;

public class DocumentCollection {
    private HashMap<String, Document> documents = new HashMap<>();

    public DocumentCollection(String[] paths) {
        try {
            for (String path : paths) {
                documents.put(path, new Document(path));
            }
        } catch (Exception e) {
            System.out.println("Error Reading file!: " + e.getMessage());
        }
    }

    /**
     * calculates tf_idf
     * @param term:     term w to calculate tf_idf
     * @param document: document d in collection to calculate tf_idf
     * @return value for tf_idf
     */
    public double get_tf_idf(String term, String document) {
        double tf = documents.get(document).get_tf(term);
        try {
            double idf = get_idf(term);
            return calculate_tf_idf(tf, idf);
        } catch (Exception e) {
            System.out.println("Term does not occur: " + e.getMessage());
            return 0;
        }
    }

    public static double calculate_tf_idf(double tf, double idf) {
        return tf * idf;
    }

    /**
     * returns idf
     *
     * @param term: given term i to use to calculate idf
     * @return value for idf
     * @throws Exception
     */
    public double get_idf(String term) throws Exception {
        int n_i = 0;
        for (Document document : documents.values()) {
            if (document.doesTermAccur(term)) {
                n_i += 1;
            }
        }
        return calculate_idf(documents.size(), n_i);
    }

    /**
     * Method to calculate inverse document frequency
     * @param N:   Total number of documents
     * @param n_i: Number of occurrences of a term
     * @return value for idf
     */
    public static double calculate_idf(double N, double n_i) {
        return log2(N / n_i);
    }

    private static double log2(double x) {
        return (Math.log(x) / Math.log(2));
    }
}
```