# Exercise Sheet 2

## Information Retrieval I

*Submit your solutions until **Monday, 07.03.2016, 12h00** by uploading them to ILIAS. Later submissions won't be considered. Every solution should contain the **name(s), email adress(es)** and **registration number(s)** of its (co-)editor(s).*

## 1 Similarity Computation (22 Points)

### 1.1 Term Weights (12 Points)

Consider a corpus of the following 4 documents.

$d_1$. **Play Games at Online Games, Games from Shockwave** Play Games on the One-and-Only Shockwave. Shockwave is the ultimate destination for free online games, download games, and more!

$d_2$. **Free Games Downloads + Free Online Games** Free games downloads and free online games. Download free full version PC games, play free online games, arcade games and puzzles. Fun and friendly community

$d_3$. **Free PC Games Download - Full Version PC Games** Download Free PC Games Download - Full Version PC Games Download. Download Free PC Games. More Free Games. FREE GAMES HOME DOWNLOAD FREE GAMES ONLINE FREE

$d_4$. **Download Free Games - Awesome Selection of Safe Game Downloads** Download Free Games has over 1000 high quality and safe game downloads and free online games. Find a fun game, download free trials, watch game video

Now, our goal is to calculate the *term weighting* of the following 8 words (separated by ";"). Assume case-insensitivity (do not distinguish between capital and lowercase letters):

- *PC; games; free; download; more; full; Shockwave; quality*

First you have to calculate the *idf* parameter (compare lecture slides). Do NOT use stemming or stopword removal. As shown in the lecture, the result of *idf* is high, if the term i (which is one of our 8 words) occurs only in very few documents. Use the following formula to calculate the *idf* weights:

$$idf_i = log\left(\frac{N}{df_i}\right)$$

where

$N$: number of documents in entire collection in the corpus

$df_i$: number of documents with term i

After you have calculated the $idf_i$ values, determine the number of occurrences $tf_{i,j}$ of all 8 terms i in all 4 documents j.

Now you are ready to calculate the weights $w_{i,j}$ for each term i and document j according to the following formula:

$$w_{i,j} = tf_{i,j} \cdot idf_i$$

Be precise up to 5 decimals and use tables to present your results. Furthermore, provide at least one example where you show HOW you used the given formulas in order to calculate your results.

Finally, comment your results. Did you find any surprisingly high or low weights? Which factors influence the calculation of the weights?

## 1.2 Cosine Similarity (10 points)

In this exercise, you have to reuse your results from above in order to calculate the cosine similarity between the documents and the following two search queries:

$q_1$: *PC games free download*

$q_2$: *Shockwave quality games*

For this purpose, use the formula from the lecture slides:

$$sim(q_j, d_k) = \frac{q_j \cdot d_k}{|q_j| \cdot |d_k|} = \frac{\sum_i q_{i,j} \cdot w_{i,k}}{\sqrt{\sum_i q_{i,j}^2} \cdot \sqrt{\sum_i w_{i,k}^2}} \quad (1)$$

where

$q_{i,j}$ = {1 if the word i is contained in query j, 0 else }

$d_k$ = Vector representation of document k (in our case one of the 4 documents from above).

$q_j$ = vector representation of query j (in our case one of the 2 queries above).

Calculate the similarities *sim* for all 4 search results and all 2 queries. As in the task before, correct to 5 decimals and use tables to present your results. Also provide at least one example where you show HOW you used the given formulas in order to calculate your results.

# 2 Evaluation: Precision, Recall & F-Measure (8 Points)

Assume that a (quite small) search space $\mathcal{D}$ contains a total of 20 documents $d$. For the query $q$ the first 10 documents are **relevant** $\{d_1, d_2, ..., d_{10}\}$, while the remaining documents $\{d_{11}, d_{12}, ..., d_{20}\}$ are assumed to be **not relevant**.

We apply three different search algorithms $a_1$, $a_2$, and $a_3$ on the total search space $\mathcal{D}$. Table 1 shows their results.

| rank | $a_1$ | $a_2$ | $a_3$ |
|---|---|---|---|
| 1 | $d_1$ | $d_1$ | $d_1$ |
| 2 | $d_2$ | $d_2$ | $d_2$ |
| 3 | $d_5$ | $d_4$ | $d_4$ |
| 4 | $d_6$ | $d_5$ | $d_5$ |
| 5 | $d_{13}$ | $d_6$ | $d_9$ |
| 6 | | $d_7$ | $d_{10}$ |
| 7 | | $d_8$ | $d_{12}$ |
| 8 | | $d_9$ | $d_{13}$ |
| 9 | | $d_{10}$ | $d_{14}$ |
| 10 | | $d_{11}$ | $d_{15}$ |
| 11 | | $d_{12}$ | $d_{20}$ |
| 12 | | $d_{13}$ | |
| 13 | | $d_{14}$ | |
| 14 | | $d_{15}$ | |
| 15 | | $d_{18}$ | |
| 16 | | $d_{20}$ | |

Table 1: Search Results

## 2.1 (4 Points)

Calculate the precision, recall and balanced F-measure ($\beta = 1$ or $\alpha = 0.5$) values for the three search algorithms $a_1$, $a_2$, and $a_3$ on the search space $\mathcal{D}$.

## 2.2 (4 Points)

Discuss your results. Explain the trade-off between precision and recall.

# 3 Search Engines (17 Points)

Lucene[1] is an open-source Java full-text search library. In this section we will look at a small example for its usage. First, read the code and description at `http://www.lucenetutorial.com/lucene-in-5-minutes.html`.

In Ilias you can find the *file lucene_ tutorial.zip* attached to exercise 2. Download and unzip it to find the required files. The file *commands.txt* contains the commands to compile and execute the program *HelloLucene* (Java must be installed).

## 3.1 Simple Queries (6 Points)

### a) (3 Points)

Try out the following queries and report the output:

- Lucene
- Action
- lucene in action
- GigaBytes
- gigabyte

### b) (3 Points)

- Why wasn't "Managing Gigabytes" found when you searched for "gigabyte"?
- Assume that the desired result of the query "gigabyte" is "Managing Gigabytes". Which type of linguistic preprocessing would have to be performed on the book titles, in order to get this result?

[1] `https://lucene.apache.org/`

### 3.2 Customized Queries (8 Points)

The search behaviour can be altered with special query operators. Compare the query pairs below: report their output and explain the differences.

- *lucene and action* vs *lucene AND action*
- *managuing* vs *managuing*$\sim$
- *a* vs *a**
- *"lucene NOT action"* vs *"lucene" NOT "action"*

*HINT:* This page might be useful for the explanations:

`https://lucene.apache.org/core/old_versioned_docs/versions/2_9_1/queryparsersyntax.html#Proximity%20Searches`

### 3.3 Search Fields (3 points)

Change the code so that the search works for the *isbn* field! Write down the resulting line of code and one example query along with its output.

# 4 Hands on (21 Bonus Points)

**The points of this exercise are bonus points. They do not influence the total amount of achivable points. You have to hand in your program code via Ilias! Please upload the compressed folder[2] instead of single files.**

The representation of documents is an important matter in information retrieval. For this reason it is worth taking a look at the preprocessing steps. In the following you will implement some simple preprocessing methods by completing the Java code template available in Ilias. Download and extract the file *ex02_source.zip*.

It contains the code template *Ex02.java* and a folder *data* where you can find the text of William Shakespeare's play *Macbeth*, its license[3] and a file with the name *english*.

## 4.1 Preparation

Inspect the code in *Ex02.java* with your favorite editor or development environment (e.g. Eclipse): Below the import statements there is the class Ex02 which is responsible for loading files, calling respective functions on them and testing the program. Point the *ex02_path* variable in the main function to the local path where you downloaded and unpacked the text data.

The class BOW stands for Bag of Words and represents a document. By solving the subtasks it will gain some document processing functionality. Right now it is not operational, because some code is missing. Note that the tasks you are meant to solve are marked as "TODO EX 4.X" with X standing for the number of the task to give you some orientation. You can remove every comment after completing the respective task. Solve them in the order given by this sheet.

## 4.2 Store & Count(10 Points)

First, you have to represent documents by means of bag of words. Basically, that is counting words. We restrict ourself to one document for now, namely *data/the_tragedy_of_macbeth.txt*.

The main method already initializes the File object and tries to build a BOW object with it to run a test afterwards.

To make it work you have to do three things:

1. introduce a datastructure to store words along with their counts
2. complete the PROCESS method to update the datastructure

[2] You can use software like 7-zip for that `http://www.7-zip.org`

[3] If you want to do anything other than solving this exercise with this data, have a look at the *licence* subfolder and `http://www.gutenberg.org/ebooks/100`

3. implement the GET_COUNT method to enable queries

The datastructure needs to **map** a **word** to the **number** of its occurrence in the text, thus retrieving the count given the word should be easy. Figure out which Java package provides the required functionality and import it. Replace every occurrence of "DATASTRUCTURE" in the code with the required type. This should allow you to run the program without any errors. The program should generate the following output:

```
'macbeth': 0
'Macbeth': 0
'MACBETH': 0
'MACBETH.': 0
'the': 0
```

The PROCESS method is called by the constructor of a BOW object. It is responsible for the word counting and already contains some code responsible for reading a file word by word. Finish this function! Keep in mind that each word either has already occurred or it is new and has to be introduced to the datastructure.

After that finish the GET_COUNT method. It needs to be able to return the counts of a given word or zero if the word is not present.

When you completed these tasks the output should look like this:

```
'macbeth': 0
'Macbeth': 30
'MACBETH': 1
'MACBETH.': 205
'the': 594
```

### 4.3 Normalization (5 Points)

As you see we could use some normalization. Finish the LOWER and the REMOVE_PUNCTUATION method.

The LOWER method returns an all lower case version of the input.

The REMOVE_PUNCTUATION method – as the name suggests – removes all punctuation symbols from a string. Maybe the REPLACEALL method of Java Strings can help you here. *Hint:* For using predefined regex character classes, the backslash needs another escaping backslash. For example the predefindex character class *digits* is represented by "$\backslash\backslash d$".

The new output is:

```
'macbeth': 284
'Macbeth': 0
'MACBETH': 0
'MACBETH.': 0
'the': 740
```

### 4.4 Stopwords (6 Points)

After normalization we can see that the output could use some cleanup because 'the' does not contribute any semantics. We should remove stopwords.

The folder *stopwords* contains a file with the name *english* that consists of one stopword per line. Your job is to complete the method *read_ stopwords* of the Ex02 class. It already has an initialized HashSet called STOPWORDS. Open the stopwords file, read it line by line and add each line (=word) each to the HashSet.

After that, you have to edit your PROCESS function again to count only the words that are **not** included in the stopwordslist.

The result:

```
'macbeth': 284
'Macbeth': 0
'MACBETH': 0
'MACBETH.': 0
'the': 0
```