LAB 3: Text processing, vector space representation, TF-IDF

1. Motivation:

The derived text (e.g., by crawling) must be processed before being used in an information retrieval system. Text processing usually involves several tasks such as: tokenization, elimination of stop words, part-of-speech tagging, lemmatization, stemming, or normalization. Some of these tasks are not trivial and have a strong influence on the system behaviour.

2. Text processing:

a. Tokenization:

Tokenization is a process by which the text is divided into single tokens – parts separated by some given set of characters such as spaces or punctuation characters. For example:

Input: Friends, Romans, Countrymen, lend me your years;

Output:

Friends	Romans	Countrymen	lend	me	your	ears

Tokenization is not always a trivial task. Different strategies can generate different tokens, and thus different results in further text processing such as indexing or querying. For example:

Input: aren’t

Possible outputs:

aren’t	
aren’t	
are	n’t
aren	t

b. Stop words:

Stop words are words that do not contain any specific information about the content of the text. These are, e.g., conjunctions or prepositions (“a”, “at”, “the”, “in”, etc.). To generate lists of stop-words, one may count the occurrences of the words in documents. It can be is assumed that these words which occur frequently in many documents do not yield any important information.

c. Lemmatization and stemming:

Lemmatization and **stemming** are tools for finding the base form of a given word.

- **Stemmers** use **heuristics** for finding the base form. As a consequence, the method may fail to find the true base form, and thus the outcome of the stemmer may be invalid. The most popular stemmer for English is Porter Stemmer.

- The examples of stemming are presented below.

SSES → SS	caresses →caress
IES→I	ponies →poni ties →ti
SS→SS	caress →caress
S →	cats →cat

- **Lemmatizers** use full-morphological analysis and dictionaries to find the lemma of the word. In practical use, replacing a stemmer by a lemmatizer does not give much gain when it comes to English language, but the results of lemmatizers are significantly better than the results of stemmers for languages with more complex morphology, e.g. Polish, German, or Spanish.

3. Vector space model

Vector space model is a model for representing documents as vectors of identifiers such as terms. Having a dictionary of terms, each term corresponds to a specific component of a vector. Each component may be considered as an axis in |S|-dimensional space where S is a dictionary (collection of all terms). Coordinate-values of a vector may be considered as **weights** of terms. The basic representations are:

a. **Binary representation:** each element is simply a 0/1 (true/false) flag that indicates whether a particular term occurs in a document or not. E.g.:

		Binary	a	be	hero	not	or	to
D1	=	To be or not to be	0	1	0	1	1	1
D2	=	Not to be a hero	1	1	1	1	0	1

b. **Bag-of-words representation:** extends binary model by specifying a number of occurrences of a term. E.g.,:

		Bag-of-words	a	be	hero	not	or	to
D1	=	To be or not to be	0	2	0	1	1	2
D2	=	Not to be a hero	1	1	1	1	0	1

c. **Term frequency (TF):** The number of occurrences is normalised by the documents length. It can be done in multiple ways. The most common way to do this is to **divide the vector by its maximal value.** E.g.,:

		TF	a	be	hero	not	or	to
D1	=	To be or not to be	0	2/2	0	1/2	1/2	2/2
D2	=	Not to be a hero	1/1	1/1	1/1	1/1	0	1/1

		TF	a	be	hero	not	or	to
D1	=	To be or not to be	0	1	0	0.5	0.5	1
D2	=	Not to be a hero	1	1	1	1	0	1

d. **TF + Inverted document frequency (TF-IDF)**

Binary, bag-of-words, and TF are "local" measures which means that other documents of a collection are irrelevant when computing a vector representation. **IDF is a "global" coefficient of a single term** and is based on the number of documents that contain such term. Let be the number of documents which contain i-th term and |D| be the number of all documents of a collection. IDF(i) is defined as: $log_{\alpha}\left(\frac{|D|}{D_i}\right)$ **The greater D_i, the lower IDF(i)..** The goal of IDF is to decrease weights of these terms that occur in many documents and to increase weights of terms which are unique.

Terms	a	be	hero	not	or	to
IDF (term)	$log_\alpha\left(\frac{2}{1}\right)$	$log_\alpha\left(\frac{2}{2}\right)$	$log_\alpha\left(\frac{2}{1}\right)$	$log_\alpha\left(\frac{2}{2}\right)$	$log_\alpha\left(\frac{2}{1}\right)$	$log_\alpha\left(\frac{2}{2}\right)$
IDF (term), $\alpha = 2$	1	0	1	0	1	0

IDF may be used within TF model as a coefficient such that TF of each term is multiplied by IDF:

$$TF\text{-}IDF\ (\mathrm{D_i}, \mathrm{t}) = \mathrm{TF}(\mathrm{D_i}, \mathrm{t}) * \mathrm{IDF}(\mathrm{t})$$

E.g.,:

		TF-IDF	a	be	hero	not	or	to
D1	=	To be or not to be	0	0	0	0	0.5	0
D2	=	To be a hero	1	0	1	0	0	0

4. **Similarity** A crucial aspect of every search engine is a selection of documents (information) to be presented to the user as the answer to the provided query q.
   a. **Cosine similarity:** When using vector space model to represent documents, a provided query q can be treated as a document and may be represented by a vector as well. To compare two vectors a cosine similarity can be computed which is a cosine of an angle between two vectors, defined as:

$$\cos(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \cdot |\vec{b}|} = \frac{\sum_i (\vec{a_i} \cdot \vec{b_i})}{\sqrt{\sum_i (\vec{a_i}^2) \sum_i (\vec{b_i}^2)}}.$$

E.g., in Figure 6 a dictionary consists of 2 terms. A query q and documents D_1 and D_2 are illustrated in the 2-dimensional vector space. To determine which of these two documents is more similar to the query, a cosine of an angle between D_1 and q as well as between D_2 and q may be computed and compared. Obviously, the angle between D_2 and q is much smaller thus the resulting cosine value is greater. Cosine = 1 for 0 degrees – the greatest similarity.

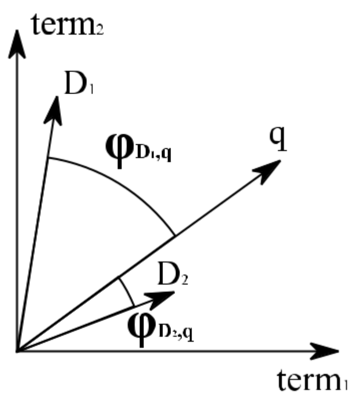

Figure 1: Cosine similarity.

b. **Jaccard index** (or a coefficient) is used for comparing the similarity and diversity of two sets. It is defined as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$

Jaccard index may be used to compare two documents if they are represented by sets. For instance:

A document may be represent by a set of unique terms it contains (special case of 1-shingling). E.g.,

			Terms (1-shingles)
q	=	I like carrots	I, like, carrots
D1	=	I like carrots very much	I, like, carrots, very, much
D2	=	I prefer carrots	I, prefer, carrots

Jaccard Index

	q	D1	D2
q	1	3/5	2/4
D1		1	2/6
D2			1

Programming assignment:

Implement simple search engine based on TF-IDF measure and cosine similarity.

1. File **keywords.txt** contains list of keywords which should be processed in your analysis.
2. File **documents.txt** contains documents for querying in the following structure:
   - first line – title,
   - following lines – content,
   - blank line – separation between documents.

   You should process title and content as one document. The code for reading documents and keywords is already implemented. You do not need to focus on that. You just get the document ready for further processing.
4. Look at the code you downloaded. **Do not change anything yet.** The template contains the following structure:
   - **Constants** (for Java in SearchEngine class, for python at the beginning of SearchEngine file).
     - **KEYWORDS_PATH –** path to file with keywords (downloaded in previous step)
     - **DOCUMENTS_PATH –** path to documents file
     - **MAX_PRINT_RESULTS –** maximal number of the most relevant documents which are displayed as a response of a query (if this number is greater than number of documents in file all documents will be displayed).
   - **SearchEngine** class - main class which is responsible for loading documents and keywords, and the workflow of search engine. You can see that its **run()** method contains the following steps:
     - Loading documents,
     - Loading dictionary (keywords),
     - Calculating IDFs for terms,
     - Calculating TFs for all documents,
     - Reading query from standard input,
     - Processing query,
     - Calculating similarity between query and each document,
     - Sorting documents based on similarity,
     - Displaying documents ranking.

     Engine keeps asking for next query until you write q in console.
     The implementation of this class is complete. You will need to fill implementation of methods which are called by method **run()**.
   - **PorterStemmer** class – contains full implementation of Porter Stemmer downloaded from https://tartarus.org/martin/PorterStemmer/ (you can find there implementation for many programming languages if you will need it in the future). It is complete. **Do not change it.**
   - **Stemmer** class – contains only one static method where you need to call proper methods from **PorterStemmer** class**.**
   - **Dictionary** class – contains 2 fields and one method:
     - **_terms** – list of unique keywords from file after stemming them,

        - **_idfs** – map key: term, value: IDF for this term
        - **calculateIdfs()** method – where you will implement filling the map **_idfs.**
    - **Document** class – class representing single document:
        - **_content –** document content before processing
        - **_title** – document title
        - **_terms –** list of processed and stemmed terms in document
        - **_tfIdfs –** vector (list) of TF-IDF values for this document. Each element of this list should correspond to TF-IDF value of term from dictionary e.g. if dictionary._terms[1] = "information", then document._tfidfs[1] = TF-IDF(document, "information")
        - **preprocessDocument() –** method which does the text preprocessing: normalization, tokenization and stemming. **Implementation is complete.**
        - **calculateRepresentations() –** method which computes vector representation (TF-IDFs) for current document. **Implementation comlete.**
        - **stemTokens()** – method which call stemmer for each token – **complete.**
        - **normalizeText()**
        - **tokenizeDocument()**
        - **calculateBagOfWords()**
        - **calculateTfs()**
        - **calculateTfIds()**
        - **calculateSimilarity()**

5. Implement text preprocessing. Fill **normalizeText()** and **tokenizeDocument()** methods in document class based on comments in code.
    - In **normalizeDocument** remove all non-alphanumeric signs (keep **letters**, **digits** and **spaces)**. You can use proper regular expression.
    - In **tokenizeDocument** split normalized document into single tokens by separating it by spaces.
    - Implement **stem** method from **Stemmer** class. Use **PorterStemmer** class to do that. You can see exemplary use of this class in main() method of PorterStemmer.
6. Implement IDF values computing in method **calculateIdfs** in **Dictionary** class. Follow comments in code. Use **natural logarithm** in order to calculate values.
7. Find TF-IDF values for document. You need to fill the following methods:
    - calculateBagOfWords – should return dictionary – key: tern value: number of occurences of this term
    - calculateTfs – should return dictionary – key: term, value: tf of this term for this document.
    - calculateTfIds – should return list of TF-IDF values for document.

8. Implement code for calculating cosine similarity between document and query (**calculateSimilarity()** in **Document** class). Follow code comments to find similarity between pair of vectors.
9. Test your search engine with some queries.
10. Verify your implementation by changing **documents.txt** and **keywords.txt** files to **documents-lab3.txt** and **keywords-lab3.txt** containing simple examples that can be computed manually.
11. Submit your python code in a single zip file, denoting the name of your team members on Spectrum