LAB 5: Query Expansion + Word Net

1. **Motivation:** The performance of every search engine strongly depends on the query provided by the user. By expressing his or her question differently, different results, e.g., rankings of the most relevant pages, may be obtained. Thus, a good search engine should aid the user in (re)formulating a query. This involves, e.g., suggesting new words like synonyms, various morphological forms, or words that frequently co-occur with the words of the provided query (see Figure 1). The search engine may also fix spelling errors ("neighborhood" → "neighbourhood"; Figure 2). Some techniques are more decision-aiding oriented. For instance, the search engine may allow adjusting weights of the words of the query. Such adjustment may be based on the selection of (ir)relevant documents (user's feedback). The goal of such techniques is to improve the retrieval process and help the user to find answers that are the most relevant to him or her.

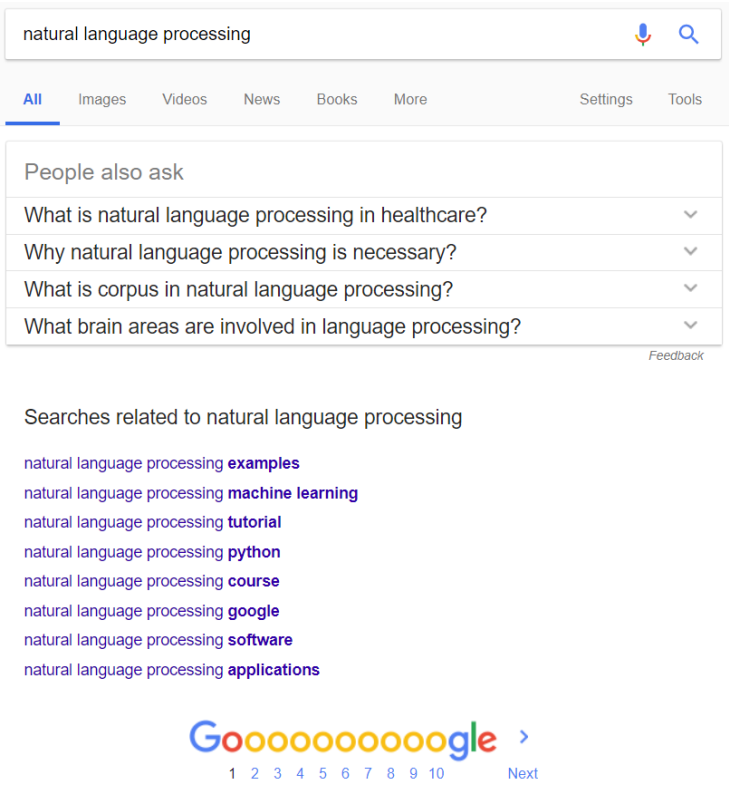

Figure 1: Google – query expansion.

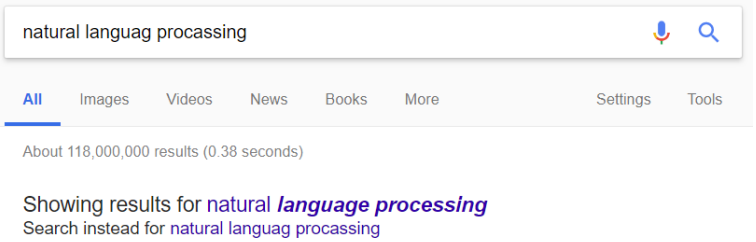

Figure 2: Fixing spelling errors.

2. **Query expansion based on dictionaries:** Query expansion techniques can use dictionaries to suggest new words that can enrich the provided query. The search engine may use a **pre-constructed dictionary** that is maintained by editors. Obviously, it is expensive to maintain such database. However, it is expected that the provided information is verified and valid (synonyms, morphological forms, etc.). An example of such dictionary is **WordNet**:

- http://wordnet.princeton.edu,
- a detailed database of semantic relationships between English words,
- developed by famous cognitive psychologist George Miller and a team at Princeton University,
- about 144,000 English words,
- nouns, adjectives, verbs, and adverbs grouped into about 109,000 synonym sets called synsets.

Another example of a dictionary is **Słowosieć (**http://plwordnet.pwr.wroc.pl/wordnet/) – a dictionary of the Polish language.

A dictionary may also be created **automatically**. It is much cheaper and faster approach than using dictionaries maintained by editors. On the other hand, the generated data may not be valid. An example of automatically generated dictionary is a **correlation matrix**. Correlation matrix C is defined as $C = AA^T$, where A is a term-document matrix ($A_{t,d}$ – number of occurrences of the term t in the document d). Consider Figure 3. There are given three documents D1-3 and three terms t1="cat", t2="milk", and t3="dog". Matrix A is a term-document matrix (e.g., "cat" occurs 2 times in D1). *A-norm* is a normalized matrix A, such that every row A_t is divided by $|A_t|$ (because of that, $C_{i,i} = 1$). *A-norm-transpose* is a transposed *A-norm matrix*. See the output matrix C. You may observe that "cat" is strongly correlated with "milk". Correlation between "dog" and "cat" is medium while there is no correlation between "milk" and "dog".

A

	D1	D2	D3
t1="cat"	2	1	4
t2="milk"	1	0	5
t3="dog"	4	4	0

|t|
4.58
5.10
5.66

A-norm (At / |t|)

	D1	D2	D3
t1="cat"	0.44	0.22	0.87
t2="milk"	0.20	0.00	0.98
t3="dog"	0.71	0.71	0.00

A-norm-transpose

	t1="cat"	t2="milk"	t3="dog"
D1	0.44	0.20	0.71
D2	0.22	0.00	0.71
D3	0.87	0.98	0.00

C = (A-norm) x (A-norm-transpose)

	t1="cat"	t2="milk"	t3="dog"
t1="cat"	*1.00*	0.94	0.46
t2="milk"	0.94	*1.00*	0.14
t3="dog"	0.46	0.14	*1.00*

Figure 3: Correlation Matrix.

3. **Relevance feedback:** it is difficult to provide an adequate query when the user does not know the collection of documents. It is easy to assess the relevance of the given document though. This motivates to build a search engine based on relevance feedback. Such system expects that the user will

provide a feedback on the system's answer. In particular, the user may choose some of the documents and mark them as relevant or irrelevant. Consider the following algorithm:

a) user provides a simple query,
b) system returns a list of documents matching given query,
c) user may mark some of the documents as relevant or irrelevant,
d) system returns list of documents based on automatically reformulated query.

Rocchio method: a method for updating a query vector – vector space model must be used to represent documents, e.g., using TF-IDF vectors. Rocchio method is based on a linear combination the following vectors:

- original query q,
- documents that are selected by the user as relevant D_r,
- documents that are selected by the user as irrelevant D_{nr}.

The new (modified) query vector q_m is obtained as follows:

$$q_m = \alpha q + \beta \frac{1}{D_r} \sum_{d_j \in D_r} d_j - \gamma \frac{1}{D_{nr}} \sum_{d_j \in D_{nr}} d_j,$$

where α, β and γ are weights. When the user assessed many documents, then values of β and γ should be higher to increase the impact of the feedback on the elaboration of the new (modified) query. Additionally, positive feedback usually has a greater influence than the negative (β > γ). Consider Figure 4. It illustrates several documents. The user selects some of them as relevant or irrelevant. The original query is depicted as well as the modified query. The weights are α =0.5, β=0.7 and γ=0.1. The β is the greatest and it can be observed that the query moved toward the centroid of relevant documents.

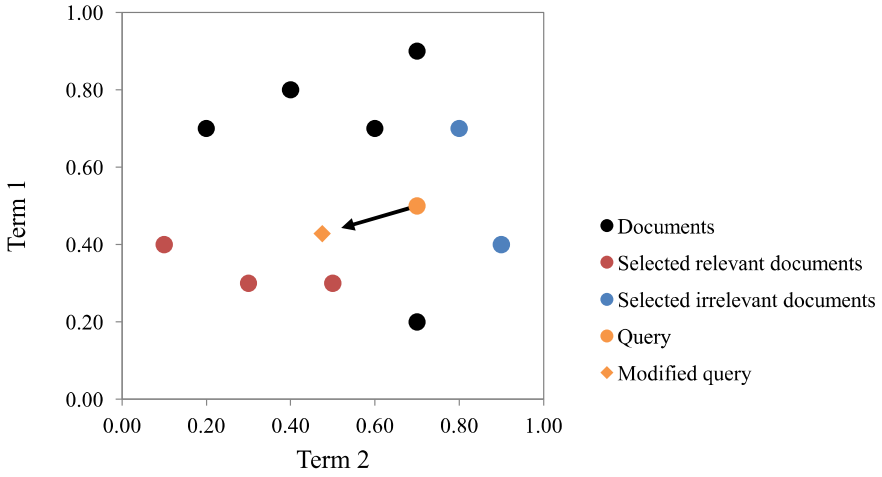

Figure 4: Rocchio method for modifying a query. *α =0.5, β=0.7 and γ=0.1.*

Programming assignment (Java, deadline +1 week)

In the following four Exercises, your task is to implement methods for query expansion. Firstly, download **project.zip.** This archive contains several java classes put in the two catalogues (packages), as well as some auxiliary files. The structure and a brief description of these classes and auxiliary files is as follows:

- **engine (package):** this package contains core java classes:
  - **Dictionary:** It keeps a list of keywords as well as terms (stemmed and normalised keywords) + IDF values of terms + terms-to-keywords map object.
  - **Document:** Represents a document. It keeps the document's title, content, ID, and terms that this document contains. Additionally, bag-of-words, and TF, TF-IDF representations of this document are stored.
  - **SearchEngine:** Main class. When run, it loads keywords, documents, computes required data (documents' representations, IDF values, etc.), and searches for the most similar (relevant) documents to the provided query.
- **search (package):** Different methods for computing a similarity between the provided query and the documents, are kept here.
  - **Score**: Keeps a pair <document, similarity between the document and the query>.
  - **ISearch**: Interface. ISearch implementation is used by SearchEngine to retrieve a sorted list of Scores. This list is sorted according to the similarity. The greater is the similarity the greater is the relevance (1^{st} position – the most relevant document).
  - **DummySimilarity (implements ISearch)**: This class does nothing. It simply returns an unsorted list of Scores. Each similarity is set to 0.
  - **CosineSimilarity (implements ISearch)**: Returns a list of Scores that are sorted according to the cosine similarity between the documents and the query.
  - **RelevanceFeedback (implements ISearch)**: It works similarly to CosineSimilarity. However, it modifies original query vector before computing the cosine similarity. For this purpose, it uses Rocchio method.
  - **QueryExpansion (implements ISearch)**: Firstly, it constructs a correlation matrix. This matrix is used then to enrich the query with some new keywords that are correlated with the query keywords. Then, cosine similarity is computed and the ranking is retrieved.
  - **WordNet (implements ISearch)**: It seeks for synonyms for the query keywords in the WordNet dictionary and presents them to the User. However, the query is not modified. The method returns a ranking according to cosine similarity.
- **keywords.txt:** contains list of keywords. Used to construct a dictionary.
- **documents.txt**: contains several documents separated by blank lines. Used to generate a list of Documents.
- **wn3.1.dict:** Folder. Contains WordNet data.
- **en-token.bin**: Apache OpenNLP model for tokenization.

Create a Java project using any Java IDE. Add the above files to your project. Then, download the following jars and add them to your project:

- extjwnl-1.9.4,
- opennlp-tools-1.8.3,
- commons-math3-3.6.1,
- log4j-1.2.17,
- slf4j-api-1.7.25,
- slf4j-log4j12-1.7.25,
- concurrentlinkedhashmap-lru-1.3.2.

Exercise 1 – search engine

(1) Go to **SearchEngine.java** and look for **TODO**. You can see several **ISearch** implementations that are commented out. Only **DummySimilarity** object is left uncommented. Run the program (**SearchEngine.java**). You can observe that:
   a. **Tokenizer** and **StemmerPorter** are initialized.
   b. Dictionary is created. For this purpose, **keywords.txt** is loaded and each keyword is parsed, i.e., tokenized, stemmed and normalized, and a list of **terms** is created.
   c. The documents are loaded from **documents.txt** and processed (list of terms is derived).
   d. IDFs values are computed and stored in **Dictionary**.
   e. Vector representation of each document is computed. It involves: bag-of-words representation, TF representation, and TF-IDF representation.
   f. Query is provided (query = "machine learning"). It is processed then in the same way as a Document object is processed. (tokenized, stemmed, normalized, vector representations are computed).
   g. Lastly, documents are sorted according to **DummySimilarity** (this class does not compute the similarity) and 10 the best results are printed.

(2) Now, your task is to use the cosine similarity to derive the most relevant documents. Switch **ISearch** to **CosineSimilarity**.

(3) Run the program. It should not work. There are several methods in the code that are not finished.

(4) Go to Dictionary.java. You can see that **computeIDFs** method is incomplete. Finish the method. Use the provided hints.

(5) Do the same for **computeVectorRepresentations** method of **Document** class and **getSortedDocuments** of the **CosineSimilarity_TF_IDF** class.

(6) Run the program. Verify the results. Do you think that the provided ranking contains relevant documents? Remember that query = "machine learning". Why are the obtained scores (cosine similarities) very small?

Exercise 2 – relevance feedback

(7) In this exercise, you are asked to implement Rocchio method. Go to **SearchEngine.java** and uncomment the proper **ISearch** object. Consider the arguments that are passed to the method's constructor. These are:

a. List of documents,
b. Alpha weight,
c. Beta weight,
d. Gamma weight,
e. List of IDs (indexes) of selected relevant documents,
f. List of IDs (indexes) of selected irrelevant documents.

(8) Leave the weights as they are. Consider a scenario in which the User selects documents **ID=7** and **ID=77** as irrelevant and selects **ID=66** and **ID=27** as relevant. Update the arguments passed to the method's constructor, i.e., (e) and (f). Check the position of these documents according to the cosine similarity (e.g., **ID=7** is the most relevant document according to the cosine similarity).

(9) Go to **RelevanceFeedback.java** and complete **getSortedDocuments** method. Follow the **TODOs**.

(10) Run the program. How does the ranking look like now? What is the rank of the documents **ID=66** and **ID=27**? Seek for **ID=7** and **ID=77**. Can you find them?

(11) Compare the (printed) TF-IDF and modified TF-IDF vectors. Can you see the difference? Why are the values of the obtained scores (similarities) very high now?

Exercise 3 – query expansion:

(12) In this exercise, you are asked to implement a method for query expansion based on automatically generated correlation matrix. Firstly, the **QueryExpansion** object must compute the correlation matrix based on the terms that are stored in the Dictionary. Then, to expand the query, the method has to find the most strongly correlated term for each term of the query, and has to add these new terms to the original query (expand the original query).

(13) Go to **SearchEngine.java** and switch **ISearch** implementation to **QueryExpansion**.

(14) Go to **QueryExpansion.java**. Complete the object's constructor where the _correlationMatrix is computed. Follow the **TODOs**.

(15) Go to **getSortedDocuments** method of **QueryExpansion.java.** Follow the **TODOs**.

(16) Run the program. Which keyword was found for "**learning**"? Which keyword was found for "**Machine**"? Does it make sense? Why? Why not? What are the correlations? Consider the modified query and the top documents. Can you see the new query keywords in these documents? Can you see the document **ID=7** in the top 10 documents?

Exercise 4 – WordNet:

(17) Your task is to use **WordNet** dictionary to get synonyms for the query keywords. However, do not use them to expand the query. Use them only to suggest new keywords that could be incorporated into the query.

(18) Go to **SearchEngine.java** and switch **ISearch** implementation to **WordNet**.

(19) Go to **getSortedDocuments** method of **WordNet.java.** Follow the **TODOs**.

(20) Run the program and verify the results. Check the results for **POS.NOUN** and **POS.VERB** parameters of the **lookupBaseForm()** method. Justify the results.