Algorithms – Homework 2

Student	Sascha Feldmann (547307)
Due Date	01-07-2015
Description	N-Gram index based search
Source Code	https://github.com/sasfeld/Algorithms/tree/master/homework2/Homework2

Overview

The application is developed by clearly separating concerns. The user has to trigger the preprocessing and N-Gram generation steps him-/herself so that he is able to do further academical researches (e.g. N-Gram search without preprocesses for a better search). It follows the UML diagram shown in figure 1.

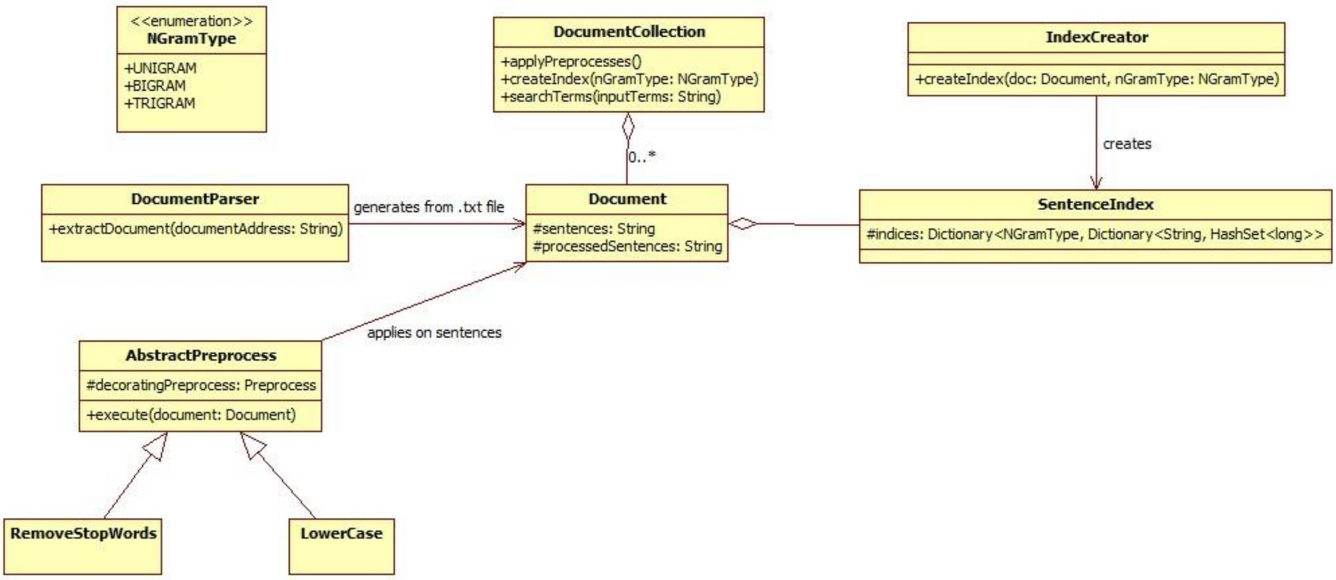

Figure 1 - UML diagram

The central "DocumentCollection" allows the search over multiple documents.

Task 1: Implementation of preprocesses

I implemented the removing of stop words and lower-casing of words which are preprocesses that take place on the parsed sentences (the raw material).

Deletion of stop words

My application uses a list of stop words that can be defined in a simple .txt – file containing stop words per line (-> figure 2). I used the "very long" (495 words) English stop word list offered by [1].

```
[This stopword list was taken from http://www.ranks.nl/stopwords]

a
able
about
above
abst
accordance
according
accordingly
across
act
actually
added
adj
affected
```

Figure 2 - Screenshot of english_stopwords.txt

The implementation is contained in the *execute()* – method of "RemoveStopWords". It iterates over each raw sentence of the given document and removes the stop words that were extracted from the txt file by using regular expression. Figure 3 shows the implementation and the regular expression. There, the English stop word is quoted into a word boundary regular expression. Also, all punctuation marks get removed.

```
foreach (String stopWord in this.englishStopwords)
{
    String regex = @"\b?\b";
    // use quoting instead of string concatenation - otherwise the pattern will b
    regex = regex.Replace("?", stopWord);

    // replace matched stopword optionally followed by whitespace by whitespace
    replaced = Regex.Replace(replaced, regex, "", RegexOptions.IgnoreCase);

    // replace sentence characters
    String regexChars = @"[;,.!?'-]+";
    replaced = Regex.Replace(replaced, regexChars, "", RegexOptions.IgnoreCase);
}
```

Figure 3 - Implementation of stop word deletion

The cost of this operation is $\log(n * m)$ while n is the number of raw sentences and m the number of stop words. For example, the removing of stop words in the Leipzig Corpora with 100,000 sentences and the long stop word list as mention above results in $495 * 10^5$ iterations.

Lower-Casing words

The application makes simple use of the String – ToLower() – method and also removes leading and trailing whitespaces.

After both preprocesses are done, the user can trigger the generation of the N-Gram index.

Let’s take a look at a concrete example.

- **Original sentence**: Asked if he might bring the world leaders to Texas, possibly to San Antonio, the president remarked, "That's a distinct possibility.
- **Preprocessed sentence:** asked bring the world leaders to texas to san antonio the president remarked "thats distinct possibility

Implementation of N-Gram index creation

The class “IndexCreator” is responsible for the N-Gram index generation. It fills the index in the instance of “SentenceIndex” that is attached to each document. The application makes use of a dictionary with N-Gram enumerations as keys and another dictionary of N-Gram String and Long set pairs. This allows flexible and separate indices for each N-Gram type (such as unigram, bigram and trigram).

When creating the index, each *processed* sentence is split into n-grams according to the N-Gram type at the punctuation characters or whitespaces. Those are our word boundaries. Afterwards, the n-grams will be added as keys if they didn’t exist in the document’s sentence index yet and been given the sentence’s number. If the keys already existed, the sentence index will be added.

This leads to redundancies of the position information that could be solved by adding subsequences for each processed sentence.

Task 2: Implementation of similarity

The task was to implement two different ways of searching sentences, one was the search by letting the user enter some search terms and finding sentences which have the largest similarity. The other was to enter a complete sentence and find sentences with a high similarity.

Implementation of Levenshtein distance function

The implementation of Levenshtein follows [2]. The algorithm is designed by the use of the dynamic programming (DP) model. The results of the Levenshtein comparison are stored in a table (two-dimensional array). Each further iteration in the algorithm makes use of previous results. So, a problem was divided into several sub-problems by identifying recurrences.

Implementation of search for terms

The easiest way to find sentences with the largest similarity is to directly use the generated N-Gram indices that were described above. Therefore, the method *searchTerms()* in the class “DocumentCollection” does the following steps:

1. Prepare a dictionary of String – Long array pairs to store search results where the string is the name of the document and the long array contains the sentence indices.
2. Iterate through all documents and delegate to their *searchTerms*() method which returns the long array of sentence positions.
3. Document::searchTerms() checks the three different indexes for unigrams, bigrams and trigrams by directly handing in the split search terms. E.g., for unigrams, the search term is split into single words and so on.

4. The similarity check has to be done from outside (the client method). The controller class therefore makes use of the Levensthein – distance function.

For example, the search for the terms “severance costs” will lead to two unigrams – “severance” and “costs” – and one bigram – “severance costs” itself. “DocumentCollection” will only return the indices matching to the two unigrams and the bigram. The client has to read the sentences using the returned indices and compare it with the search terms itself using the Levensthein distance.

Implementation of search for similar sentences

The class “DocumentCollection” also offers a method *searchSimilarSentences()* which expects the parameters *sentence* and *maxDistance*. The user can type in the maximum Levenshtein distance he would like to have between his input sentence and those contained in the corpus.

The algorithm works as following:

1. Prepare a dictionary with the searched document as key and a sorted dictionary as value. That inner sorted dictionary consists of integer and HashSet<long> pairs where the integer is the Levenshtein distance and the set contains the sentence indices.
2. Same as for search of terms (see above), but instead of terms the complete sentence is given to Document::searchTerms().
3. Document::searchTerms() will extract all uni-, bi- and trigrams from the input sentence and check the document’s dictionaries for sentences that contain those N-grams.
4. Calculate the Levenshtein distance for each returned sentence.
5. If the distance is less than the given maximum of the user, append the list of similar sentences to the resulting list.

ngrams	{string[19]}	string[]
[0]	"Asked"	string
[1]	"if"	string
[2]	"he"	string
[3]	"might"	string
[4]	"bring"	string
[5]	"the"	string
[6]	"world"	string
[7]	"leaders"	string
[8]	"to"	string
[9]	"Texas"	string
[10]	"possibly"	string
[11]	"San"	string
[12]	"Antonio"	string
[13]	"president"	string
[14]	"remarked"	string
[15]	"\"That's"	string
[16]	"a"	string
[17]	"distinct"	string
[18]	"possibility"	string

Figure 4 - Generated unigrams for example sentence (Visual Studio Debug view)

Figure 4 shows the unigrams that are identified by Document::searchTerms() for the user input of “Asked if he might bring the world leaders to Texas, possibly to San Antonio, the president remarked, "That's a distinct possibility.”. All sentences that contain one of those unigrams will be returned. Then, the distance check will be done between the input sentence and the returned ones.

Growth rate

First, let’s split the work that has to be done in order to search for terms and similar sentences. Both consist of the following steps:

- Generate N-Grams for the user’s input (complete sentence or single terms)
- Search sentences through documents using the dictionaries / maps
- Compute IDs of mapped sentences
- Apply Levenshtein function on the user’s input and the returned sentences

Generation of N-Grams

The generation of the three different N-Grams depends on the size N that is the number of words within the search term or sentence.

If you search for “president NASA USA shuttle” (problem size = 4) for example, the terms will be split into 4 unigrams, 3 bigrams and 2 trigrams:

- Unigrams: president, NASA, USA, shuttle (= N)
- Bigrams: president NASA, NASA USA, USA shuttle (= N-1)
- Trigrams: president NASA USA, NASA USA shuttle (= N-2)

Because trigrams are currently the maximum, the overall runtime can be rounded to N.

Search of sentences

For the look-up for the sentence indices to the given N-Gram is done using hash maps, so the runtime is $\log(m)$, where m is the number of indexed N-Grams.

$$O(m) = \log(m)$$

Compute IDs of mapped sentences

The hash map of the previous step returns a HashSet of long values that represent the sentence IDs. The worst case would be that all sentences of the index are contained in this set if the index consisted of identical sentences. So, let’s call the growth rate here K.

$$O(k) = k$$

Levenshtein distance

The calculation of the Levenshtein distance is done by a matrix where the columns are the input search terms or the complete sentences and the rows are the sentences that were looked-up (using the N-Grams above). So the runtime is $O(N * k)$, where N is the number of words within the user's input (see above) and K is the number of terms within the looked-up sentence.

$$O(L) = N * k$$

Overall growth rate

So the Levenshtein distance seems to be the most expensive operation for the similarity check. It depends on the number of words within the looked-up sentences or the user's input, so there's a high risk to cause a large runtime.

For the search of terms itself, it doesn't really seem necessary to apply the Levenshtein distance function because the split of the search terms into N-Grams and look-up using a hash-map already ends in having sentences containing the word. Here, you could evaluate the search results by the N-Grams themselves: if a trigram such as "to San Antonio" was mapped to a sentence, than it can be seen as more similar as trivial unigram matches. The word "to" can appear in a lot of sentences, for example.

$$O(O) = O(m) * O(k) * O(L)$$

$$= \log(m) * k^2 * N$$

$$\text{O (O)} = N^3$$

N^3 can be the worst case if the index consists of identical sentences.

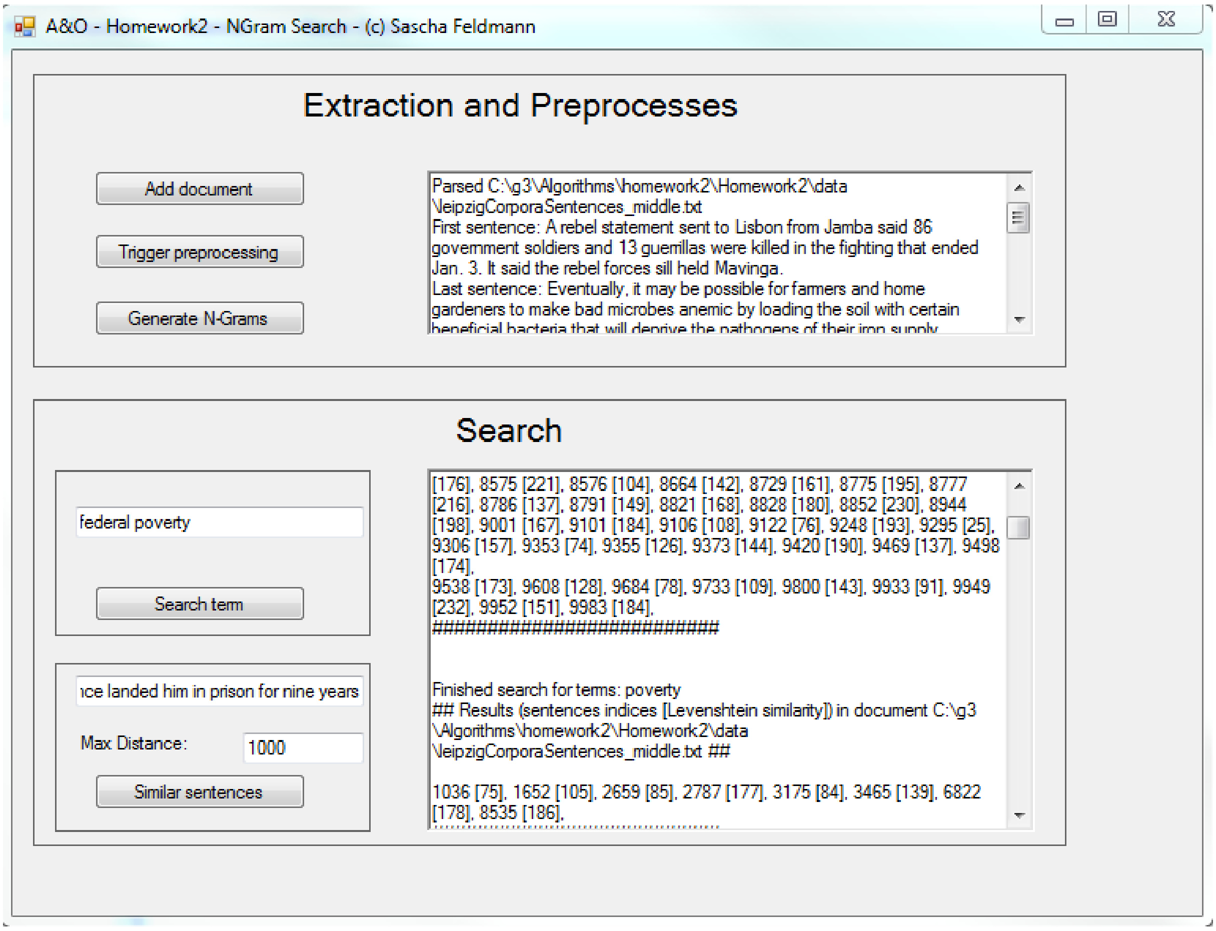

Figure 5 - Screenshot of application

Sources

[1] http://www.ranks.nl/stopwords, accessed on 12-15-2014.

[2] http://www.dotnetperls.com/levenshtein, accessed on 05-01-2014.