Wikipedia Information

Linus Hansson
Amelia Andersson
Per Classon
Anton Warnhag
Johan Ekström

May 14, 2014

Abstract

This is a skeleton for KTH theses. More documentation regarding the KTH thesis class file can be found in the package documentation.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Mauris purus. Fusce tempor. Nulla facilisi. Sed at turpis. Phasellus eu ipsum. Nam porttitor laoreet nulla. Phasellus massa massa, auctor rutrum, vehicula ut, porttitor a, massa. Pellentesque fringilla. Duis nibh risus, venenatis ac, tempor sed, vestibulum at, tellus. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos.

Contents

1 Introduction

1.1 Background

Today, information is becoming abundant. There is a vast amount of information available on the Internet and this amount is only growing in size. Because of the expanding set of relevant data, it is becoming more and more important to construct relevant overviews, which saves one from sifting through a complete corpus in order to get a general idea of a topic. Another problem may arise from the expansion of information itself; as it grows, previous summaries may become obsolete. Information abstraction is therefore an important problem to solve.

Automatic text summarization is a way of solving this problem. The idea is that texts have inherent structures that can be exploited in order to find relevant data. Different methods such as abstraction (which attempts to summarize the text in natural language) and extraction (which returns important sentences from the text itself) may be employed, either by themselves or in conjunction.

1.2 Problem

The aim of this project is to build a tool that takes a search string from a user and gives context information about the query subject back to the user. The information, taking the form of the top most informative sentences about the subject, was to come from Wikipedia.

2 Previous/Related work

There are many different algorithms one can use to automatically summarize a text. Before finally deciding to use Continuous LexRank (see chapter 2.2.1) we spent quite a lot of time looking into some of the algorithms. The results of our study are presented in the following subchapters.

2.1 Introduction: Extractive vs. Abstractive

In the field of summarization one differs between extractive text summarization and abstractive text summarization.

Extractive text summarization produces a summary of a text cluster by choosing a subset of the sentences in the cluster. The way to choose sentences may differ between methods, but each method should have some way of giving each sentence a rank to be able to sort them according to relevance.

Abstractive text summarization on the other hand involves paraphrasing sections of the cluster. By definition it involves generating text not necessarily appearing in the original documents. This generation involves abstracting words used to broaden themes and synonyms of the words in the texts. This part of the field of summarization is much less developed than extractive methods, mostly because it involves another field still in its early stages: that of natural language generation. NLG is the process of turning knowledge base information into natural language that is more easily understandable to humans.

2.2 Extractive text summarization methods

The following subsections give two main examples of different extractive text summarization methods and how they are used: Lexical Chains and LexRank (which we ended up using).

2.2.1 Lexical chains

Capturing the key points of a text is essential in automatic text summarization. A rather sophisticated approach to this is to use lexical chains. A lexical chain can be thought of as a list of words, most commonly nouns, absorbed from a document. These are meant to form a representation of the key themes of the text.

Lexical chains exploit lexical cohesion between sentences. Lexical cohesion involves the selection of a lexical item that is in some way related to one occurring previously. Repetition, i.e several occurrences of identical words is one form of lexical cohesion, but cohesion also includes synonyms, near synonyms or hierarchical connections. When cohesive elements occur over a number of sentences a cohesive chain is formed.

Implementing a lexical chains algorithm requires a database of words and their connections to other words. A resource that is commonly used to this end is WordNet. WordNet lists all the related words to any given word together with the relation. Given that only nouns are treated a Part-of-Speech tagger is also needed to single out the nouns of the text. These nouns are then grouped into chains according to the cohesive relations mentioned above. In the end the resulting chains are ranked according to strength. Strength can be determined using a scoring system depending on the length of the chain and the relationships it holds. These will then reveal the central themes of the text and a summary can be formulated. [**?**] [**?**]

2.2.2 LexRank

Degree Centrality

Degree Centrality uses the fact that many sentences in a cluster of related documents are expected to be somewhat similar to each other. The way in which Degree Centrality uses this fact is by viewing the text cluster as an undirected graph. Firstly, words are given a tf-idf score and the similarity between each pair of sentences are calculated using a idf-modified-cosine score, which is a common scoring system in Information Retrieval. Each sentence in the cluster is then represented by a node. Two sentence nodes are only connected if their similarity score exceeds a certain threshold. With this graph representation, a simple way to assess the centrality of each sentence is to count the number of similar sentences, i.e the number of neighbors the node has. This algorithm sorts all sentences according to their centrality score, in descending order, and a summary can be chosen as the k highest ranking sentences.

The Degree Centrality algorithm is interested in all similar pairs of sentences, and does not take into account the actual value of the similarity as long as they are above a certain threshold. In the graph view described above, it only counts the number of sentences that are similar according to the threshold and thus in effect removing the edges with lower similarity scores. The choice of the threshold can greatly influence the result since a high threshold will eliminate almost all similarities and a low threshold might take the weak similarities into account as well. [**?**]

LexRank

LexRank is an extension of Degree Centrality. In Degree Centrality, all edges are treated the same way, the value of the similarities are not taken into account. The idea behind LexRank is that not all edges are equally important. In Degree Centrality, if several irrelevant sentences are similar to each other, those sentences will have a high centrality score due to the fact that they are neighbors in the centrality graph. The LexRank extension tries to eliminate this unwanted behaviour.

The way LexRank does this is to use the centrality of the neighboring sentences. Each sentence has a specific centrality value, p, and shares an equal portion of this value to all its neighbors. This can be formulated as the following equation:

$$p(u) = \sum_{v \in adj(u)} \frac{p(v)}{deg(v)} \tag{2.1}$$

where p(s) is the centrality of sentence s, deg(s) is the number of neighbors of s and adj[s] is the set of neighbors of s. This equation is however unsolvable, but Erkan and Radev suggests a solution to this equation using the PageRank algorithm (hence the name LexRank, Lexical PageRank) and using some form of iterative method in order to get convergence, for example Power Iteration.

$$p(u) = \frac{d}{N} + (1-d) \cdot \sum_{v \in adj(u)} \frac{p(v)}{deg(v)} \tag{2.2}$$

where d is the possibility to perform a random jump between the sentences and N is the total number of sentences. [?]

Continuous LexRank

Neither Degree Centrality nor LexRank take the similarity weight between sentences into account, only how many similar sentences there are. The difference between LexRank and Continuous LexRank is that the cosine score is used directly in the PageRank algorithm. The graph will be denser but weighted.

$$p(u) = \frac{d}{N} + (1-d) \cdot \sum_{v \in adj(u)} \frac{idf-modified-cosine(u,v)}{\sum_{z \in adj(v)} idf-modified-cosine(z,v)} p(v) \tag{2.3}$$

These two variants of LexRank might also differ in the aspect of what counts as similar sentences. In Continuous LexRank the threshold mentioned earlier is removed, and all values of similarity are accepted. [?]

2.3 Abstractive text summarization methods

2.3.1 Opinosis

Opinosis is a concept intended to condense what the authors call highly redundant opinions. Specifically this means that it is designed to summarize comments on products of different kinds, such as you might find in below a product page in an online store or beneath a review of a movie. The idea is that the method creates a summary that includes the most repeated parts.

It does this by generating a graph of the words used, where every node stands for some specific word and points to all the other word-nodes that its word closely precedes (within a few words) in the chosen collection of opinions, along with how many times it comes up preceding that word. So this graph tells you which strings of words come up most frequently in the collection. Given some lower limit for how many times a string needs to show up to be considered, those that do are then used to construct a summary using some rather simple concepts from natural language generation, NLG, concerning the structure of sentences. In effect, the algorithm creates a rough intersect of the collections sentences, leaving out the parts that only show up in individual sentences or few.

Opinosis is not truly abstract (the authors calls it shallowly abstract) because it does not generate words that are not in the subject collection. But it does create sentences that were not in that collection, and sidesteps many of the difficulties with NLG by utilizing its sentence structure. [**?**]

3 Method

Our idea at first was to combine the concept of a word graph, as used in Opinosis, with the generation of the most important sentences from an extractive method such as LexRank. But because LexRank does not generate the highly redundant collection of sentences needed by Opinosis, we would modify it from doing an intersection to instead doing more of a union. Using a word graph we would identify parts of sentences returned that also figured in other sentences, for example names, and fuse the other parts of the sentences based around these.

However, such fusing turned out to be very difficult if we intended to return something that could compete in legibility and completeness of the summary compared with simply putting the most important sentences returned directly by LexRank in line after each other. It proved outside the scope of how much time we could spend on this project, and due to this we ended up implementing just Continuous LexRank.

We also planned to build a small and simple GUI in which a user could define her search strings. A GUI was not specified in the project description, but we considered it an important addition to our program as it was meant for human use.

3.1 What we did

We implemented the LexRank algorithm in Java and implemented it in such a way that we could change different parameters in-between runs (these parameters are described in chapter 4, Evaluation). Due to Wikipedia articles tending to have a summary early on, we also added a parameter by which we could control how early in an article a sentence had to show up to be considered to be displayed as a summary. In addition to this, we also built a small GUI in which a user can define her search strings and see the results sent back from the program, i.e. the programs summarization of the search topic.

Between our LexRank algorithm and our GUI, we implemented a processing layer. The task of this layer was to convert the search string from the user to an Apache Solr url and then to decode the result from Solr. When performing a search in an Apache Solr database, one can choose between a few output formats of the result. We chose to receive the output in the easily parsed JSON format (Javascript Object Notation) and use the JSON-simple library to decode the result and obtain the sentences from the Wikipedia articles. From the Wikipedia articles you can gather a lot of information about revisions and user edits, but we were only interested in the title and text of the current revision of the articles and therefore only extracted these.

All sentences then continued to the LexRank implementation, which ranked the sentences according to relevance to the query. The highest ranked sentences from the LexRank algorithm were then sent to back to the GUI along with the titles of the Wikipedia article containing the sentence.

3.2 Resources

The following subsections list the resources we used for our implementation.

3.2.1 Wikipedia database

Due to the massive size difference between english Wikipedia and its corresponding swedish set, we decided to use swedish Wikipedia as our source of data. Wikipedia provides their data for download in the format of XML (Extensible Markup Language). This is the only way to get all of Wikipedias articles, as Wikipedia does not want anyone to crawl their website in order to minimize traffic load. By using the database Apache Solr, it was then possible to import the XML-data and save the articles in a searchable format. As Wikipedia contains a lot of unwanted articles, such as redirects, portals, templates and categories; a regular expression pattern was used to skip those documents. [**?**]

3.2.2 Apache Solr

Solr is an open source search platform with features such as full-text search which is very scalable. It is written in Java and runs as a standalone server. It comes with a REST-api (Representational state transfer) where you can search by doing simple GET-requests. JSON was used as the result format for our searches, which was then parsed and used in our other algorithms. [**?**]

When doing full-text searches in all of the Wikipedia articles, Solr use by default a tf-idf scoring method with some modifications for ranking the articles. They have made what they call a Lucene's Practical Scoring Function, which it is based on the Vector Space Model (VSM). It uses td-idf scores in combination with document boosts (which is predefined scores of documents that should be ranked higher) and query boosts (some words in queries can be given higher priority). [**?**]

3.2.3 JavaFX

JavaFX is a GUI library for Java and a software platform for creating and delivering rich internet applications (RIAs). We used it to create a desktop application from which a wikipedia search could be done. [?]

3.2.4 JSON-simple

JSON-simple is a simple Java toolkit for encoding or decoding JSON text. We used it in order to decode the result from a Apache Solr query. [?]

4 Evaluation

To measure the efficiency of our program and algorithms we constructed an experiment where we calculated the precision of a set of 16 predefined queries (see appendix A). For each query we evaluated the summary we got back and graded it in a three graded scale as shown in Table 4.1 below.

Grade	Description
0	Not relevant
1	Relevant information
2	Relevant description

Table 4.1: The grades used to evaluate our queries.

The program had four different parameters that were changed during evaluation, and all queries were tested with each parameter setting. The parameters we used were:

1. Maximum number of documents returned from Solr.
2. Minimum number of sentences required in a document.
3. Minimum number of words in a sentence.
4. Maximum position of a sentence in a document.

For the first parameter (maximum number of documents) we tested the values one and five. With this parameter set to one, we only used the top ranked document from Solr in our summarization, and with the parameter set to five, we used the top five documents from Solr in the summarization. We could not entirely remove all unwanted articles (mentioned in chapter 3.2.1) when we imported the Wikipedia XML file to Solr. As a consequence to this, we used a guard in our implementation to skip documents of those types when doing the summarization.

For the second parameter we used the values one and ten. This parameter had in effect that we excluded documents containing less than one (or ten) sentences. Since a lot of Wikipedia articles only references other articles or are not completed, we did not want our summarization to be built on these articles.

When parsing a Wikipedia article we could end up with very short sentences, namely words that was in fact a link, a header, or something else entirely. To see how much these sentences would affect the result we used the third parameter to skip some of these. The values we used were zero and five. With zero as the limit, we accepted all sentences, even the ones described here and with five as the limit, we skipped these sentences.

The last parameter told our program where in the article the summarization was to be extracted from. We run LexRank on the entire document, but when selecting the summary we wanted to be able to use the general layout of a Wikipedia article where there are somewhat of a summary in the beginning. The values we tested was five, ten and one thousand. The value one thousand can be interpreted as the entire document since most articles contain less than one thousand sentences.

5 Results

The maximum score one could get in an evaluation was 32, indicating that a summary of the query subject was returned for every query. The average we got across all different permutations of parameters values was 15.375. The best combination of parameter values was 5, 1, 5 and 5 (the order of these values are as described in the evaluation) which gave a score of 15. For the different parameters in our evaluation we calculated some average scores:

TODO: en stor tabell

Of course, these parameters do of course work in combination, and changing one affects the others. Still, these averages should give some indication of the relative importance of the variables different values.

6 Discussion

It is clear that we can better the results significantly by taking the structure of Wikipedia into account. We can see this starkly when looking at the effects of only considering the first five or ten sentences as possible summaries, compared to considering sentences from the whole documents. Considering how much higher the score is especially when looking for summaries only among the first five of a documents sentences, there is certainly some reason to doubt the use of LexRank for Wikipedia unless you have very high demands for the brevity of the summary. With a little more relaxed limits for how long a summary can be (a single sentence is quite harsh), it may be better to just lift the five first sentences of the article directly. Indeed, we can see that this is what Google does for some searches.

This was not the only way we took the structure of our source data collection into consideration; we also tried to filter out very short pages since we thought the scores sentences in these would get or give during LexRank would not reflect well on the actual importance of sentences. We used the length of the articles as a rough measure of their exhaustiveness and authority on a subject to mitigate that TF-IDF (which Solr uses for searches) gives higher scores to shorter pages. This may have been too hasty, however, as can be seen by the results of the two different values for the minimum amount of sentences needed in a document for it to be considered: We actually got better results from having no lower limit.

Appendices

A queries

B results