Summary of Project – Text Relate

Abstract

The idea of our project is to give people a chance to read unbiased news, with different perspectives that can be found throughout the web. Instead of reading only one article, from one specific site, we wanted to create an opportunity for readers to be able to read numerous opinions about the exact same subject they were interested about.

Introduction

The input to our project is a news article. Using advanced Natural Language Processing techniques, we analyze the article and extract its main topics. From these topics we construct several queries, which we then send to Bing News Search Engine in order to find related articles. Then, we rank all the results according to the similarity between them and the input article, such that articles that talk about the exact same subject as the input article get better rankings than those who don't.

After finding the most relevant articles, we integrate with another project to produce the final result. The other project takes several similar articles as input and produces a summarization of the articles, while keeping good flow of the summarization. Combining the two projects together, from a single article, probably biased and very subjective, we create a completely new article, enriched with opinions and diverse influences.

In the next few pages we explain our approach thoroughly, starting from analyzing the input articles, and finally producing the summarization.

Extracting the main topics of an article

In order to find the main topics of the article, we decided to use the following approach: first, give a score to each word. Then, find the noun phrases with the best cumulative score. They will construct the queries in the next stage.

The scoring of the words was also divided into two parts. In the first one, we wanted to determine for each word if it is important or not. For that purpose we used the LLR (log likelihood ratio; see appendix) technique, that gives each word a score of 1 if it is important, and 0 if it is not. Moreover, we identified the named entities in the article (People, Organizations, Locations) and marked them as important, regardless of their LLR score. In the second part, we gave each word that was marked as important a score of 1+log(frequency(word)), such that words that appear many times in the article are given higher scores, but only by a logarithmic scale, and not linearly.

To be sure we didn't give common words like "the" a score of 1 (also known as 'stopwords'), we gave a score of 0 to every word in a stopword list (there are many such lists, we used the one provided by NLTK). This is because these words are obviously not significant to the topic of any article, but their frequency is high, so it is best to give them a score of 0 in case the LLR score gave them a score of 1.

Next, we searched for noun phrases in the article, and gave each one a score calculated as follows: $\frac{\sum_{word \in noun\ phrase} score(word)}{\max(log(length(noun\ phrase)),1)}$. This way a noun phrase is penalized if it is too long, but not too significantly.

Constructing the queries out of the main topics

Given the most important noun phrases in an article, we built several queries for Bing Search. We chose the queries to be as follows: the top 3 noun phrases, all combinations of pairs of the top 3 noun phrases, and all of the top 3 noun phrases concatenated. Seven queries in total. A problem we encountered with this approach was that sometimes the top noun phrases were very similar, as there are often multiple noun phrases containing the same keywords in an article, and choosing two noun phrases with similar keywords gives no additional information. We solved that problem by penalizing each word in a noun phrase chosen to be one of the top noun phrases, such that the other top noun phrases are very unlikely to be talking about the same thing.

Crawling the web

Crawling is a serious bottleneck in every project involving the web, so we multithreaded our crawler such that the crawling time was cut by no less than 8 seconds.

Ranking the most similar articles

Phase 1

We use the queries constructed earlier in order to search for similar articles via Bing Search. For each query we get about 20 articles, therefore we could get almost 150 articles in total. Surely many won't be speaking on the same subject, and some may be repeating themselves because of the combinations of the noun phrases in the queries. Analyzing each one of the 150 articles could take a long time. Therefore, we apply a quick filter on the articles and eventually analyze only about 20-30 articles.

We considered the following features for the filter:

- The number of queries that reached each article. For example, if an article A appeared in the search results of exactly two different queries, its number would be 2. If an article appeared in the search results of multiple queries, it is very likely that it has the same subjects as the input article.
- Cosine similarity between the description of the found articles and the original article.
- Cosine similarity between the title of the found articles and the original article.

It is important to emphasize that this filter is not an integral part of the project, but only a performance optimization. Our algorithm doesn't make any other use of the articles' titles and descriptions, and only takes into account the body of the articles, thus making the results much more interesting.

Phase 2

After the filter we are left with about 20 articles. In the first phase we did not analyze any of the articles at all, as mentioned earlier. The filter was based only on the knowledge we retrieved from Bing.

Each word in the article gets a score by the tf-idf technique (see appendix). As done earlier, stopwords automatically got the score 0.

Having given each word a score, we use a formula that calculates how similar two articles are, as follows:

$$\frac{\sum \mathrm{s_i} \cdot (1 - \sqrt[2]{\prod_{s_j}(1 - similar(s_i, s_j)})}{\sum \mathrm{s_i}}$$

s_i represents a sentence in the input article, while s_j represents a sentence in the second article, which is the one which we check how similar it is to the input article. The value of $\mathrm{s_i}$ (which is a weight given to each sentence in the input article) is calculated by the sum of the tf-idf values of only the important words in it, which means the LLR gave them a score of 1.

Similar ($\mathbf{s_i}$, $\mathbf{s_j}$) calculates how similar the two sentences are, by a classifier we've built. For explanation about the classifier please see the appendix.

The idea of $\prod_{s_j}(1 - similar(s_i, s_j)$ is to calculate the probability that not even one of the sentences in the second article is similar to $\mathrm{s_i}$. We took the root of it to "punish" long documents – too many sentences would get the result low even if none of them is similar to $\mathrm{s_i}$. We used the square root instead of the nth root because n is too strong, and the result would be almost always 1. Once you deduct this result from 1, you get the 'probability' of $\mathrm{s_i}$ being covered in the second article, meaning the article expresses a similar idea to that of that particular sentence. There is an option to include also the title as a sentence, to be set on or off by an input.

Another equation we tried is: $\frac{\sum \mathrm{s_i} \cdot Max_{s_j}(similar(s_i,s_j))}{\sum s_i}$, but its performance was not as good as the one above, although we kept the option to choose between the two.

The final ranking of the articles is according to this score. As close it is to 1, the higher the rank would be. The top 5 articles were sent as input to the summarizing application, and the produced summary is the product of out project.

Creating demo of our project

We've created a site based on "WordPress", which is an easy to use blog site. We update the database in which the site stores its data, including the articles, such that the articles produced by our algorithm appear on the site. This is done automatically by our program, but we can also edit the website manually as we please.

The work flow is as follows: The client gives an RSS link to our project. The RSS link holds several links of different articles. Each article is processed with our project, and yields the 5 most similar articles found. Using the second project, these articles construct a new summarized article, and that new article is then inserted into the database. To conclude, from a single RSS link you see on the site several new articles, each one related to a different article.

Appendix

LLR technique in a brief

In general, the LLR technique compares tests two hypotheses: the first one is that two distributions have the same underlying parameter, and the second is that they don't. We have used this test to determine the significance of a word in an article, by using a big unigram corpus (as taught by Jurafsky; see below). Intuitively speaking, we test the probability of a word given a big corpus and the probability of the word given an article. If the probabilities are equal most chances the word isn't unique, and if the probabilities are significantly different then it might mean that the word is important in the article. For example, if the word "Trump" is seen an article multiple times, in contrast to the corpus, where it doesn't appear much, then the probabilities would probably be very different. Therefore, chances are that "Trump" will get a score of 1. As opposed to the word "Trump", the word "the" will probably have the same probability in the article and in the corpus, and therefore will get a score of 0.

In our work, the corpus we used is taken from *Google Web Trillion Word Corpus,* specifically: count_1w.txt.

The original paper that introduced LLR, as well as the formula that we used, can be found at http://aclweb.org/anthology/J93-1003 (Dunning, 1993).

The threshold we used is taken from Jurafsky & Martin, 2nd edition, formula 23.26.

tf-idf technique in a brief

The technique of tf-idf is divided into two. The first part, tf (term frequency), gives a score to a word in an article according to its frequency in the article. If the word "Trump" appears several times while the word "Obama" appears only once, most likely the article focuses more on Trump rather than on Obama, and therefore "Trump" is a more important word than "Obama" is in this particular article.

The other part is the idf (inverse document frequency). While the tf measurement is based only on a single article, the idf measurement is dependent on many articles. Say we have about 20 articles in total and not just one. So if the word "Trump" is seen in all of them – it will get a low score, because all the articles talks about him. On the other hand, if the word "China" is seen only in two articles out of the 20, there is a good chance that they talk about the same subjects, therefore "China" will get a higher idf score than "Trump".

Eventually the tf-idf score of a word is calculated by a combination of the two scores, the tf score and the idf score.

The formula we used for the tf-idf value of a term t in a document d is:

$$\left(1 + \log tf_{t,d}\right) * \log \frac{N}{df_t}$$

Which, in words, is 1 plus the log of the number of times the term appears in the document, times the log of the number of document divided by the number of documents the term appeared in.

Classifier of similar sentences

As mentioned before, the classifier's target is to predict how similar two sentences are. Therefore we needed to give the classifier features. Each sentence has 6 different vectors representing it. One is tf-idf score of all the words in it, second is the tf-idf score of words that are part of noun phrases only, and third is tf-idf score of words that are name entities only. The three other vectors are tf scores of the same as above. The features given to the classifier are the cosine similarities between each of the six vectors accordingly.

In order to train the classifier, we've created 4 new articles, three of them talk about the same subject, and the fourth is unrelated to that subject, trying to simulate how Bing would give us articles from a query. We clustered sentences out of the four articles that approximately talk about the same things together. This way the classifier can know which two sentences talk about the same things and which don't.

We then ran a Naïve Base learning algorithm to learn when two sentences are talking about the same subject.

We are aware that in order to build a state-of-the-art classifier you need hundreds of thousands of examples, but unfortunately it was out of our reach. Nevertheless, it works quite well ☺.

Main libraries and scholarly literature

Libraries

- Stanford Core NLP- analyzing text
- OpenNLP - analyzing text

- BoilerPipe – getting the article body from a URL
- Weka – creating the classifier

Scholarly literature

Most of our knowledge in Natural Language Processing was acquired through Dan Jurafsky & Chris Manning's course in NLP, as well as their corresponding books.

Comments about the code

For the completeness of the project, we are specifying the exact libraries and versions that we used for the project:

- apache-opennlp-1.6.0
- boilerpipe-1.2.0
- httpcomponents-client-4.5.2
- mysql-connector-java-5.1.41
- stanford-corenlp-3.7.0
- apache-commons-lang
- com.google.guava_1.6.0
- gson-2.6.2
- json-20160212
- jsoup-1.10.1
- weka-3.7.3

For the WordPress site, we used Wamp. A complete tutorial on how to install Wamp and WordPress on your computer can be found here: http://www.wpbeginner.com/wp-tutorials/how-to-install-wordpress-on-your-windows-computer-using-wamp/