Information Retrieval Project: Query Suggestions

Thora Daneyko

March 9, 2017

1 Introduction

When entering a query into a search engine, it is often difficult to formulate it in such a way that it is informative for the search engine about the user's information need. Because of this, many search engines suggest likely query terms or phrases depending on what the user has entered so far in order to aid him expressing his information need in a helpful way. Many search engines rely on a large amount of query logs, i.e. queries that have been submitted before, and can deduct the most likely queries from this data. However, newer search engines or information retrieval systems that do not receive as many queries as web search engines, cannot rely on a sufficiently large query log corpus and may never be able to collect enough data from their users to provide reasonable query suggestions.

n-grams extracted from large text corpora can be a good indicator of which words frequently occur together and can thus be used to supplement or replace query logs as a basis for query suggestions when there would otherwise not be enough data to generate proper results. Bhatia, Majumdar, and Mitra 2011 have implemented such a query suggestion system based on n-gram data and have shown that it can produce high quality results. However, they used very specific corpora as a basis and only tested it on queries surrounding the same topics covered by these corpora.

In this report, I discuss my implementation of an n-gram based query suggestion system mostly following the probabilistic approach proposed by Bhatia, Majumdar, and Mitra 2011. The data underlying my system is taken from a large Wikipedia corpus, thus covering a broad variety of topics. It will be interesting to see whether it can generate meaningful query suggestions for all kinds of information needs or whether the specificity of Bhatia, Majumdar, and Mitra 2011's data is crucial to its success.

2 Query suggestions without query logs

The query suggestion system designed by Bhatia, Majumdar, and Mitra 2011, from now on referred to as BMM2011, selects phrases from a large pool of n-grams (with $n = 1$, 2 and 3) with a probability based on the phrase's frequency and its correlation with already entered query terms.

First, in case the last query term Q_t of the query Q has not been typed in completely yet, all possible completions c, i.e. unigrams starting with the incomplete query term, are retrieved. Then, the system collects all phrases p, i.e. bigrams and trigrams, containing one of the completions (or the complete last query term). Each of these phrases is assigned a probability $P(p_i|Q)$:

$$P(p_i|Q) = P(p_i|Q_t) \times P(Q_c|p_i) \tag{1}$$

The first part of this equation is the *phrase selection probability* $P(p_i|Q_t)$, i.e. the probability that the phrase is selected given the last query term:

$$P(p_{ij}|Q_t) = P(c_i|Q_t) \times P(p_{ij}|c_i) \tag{2}$$

The phrase selection probability itself is composed of the *term completion probability* $P(c_i|Q_t)$, the probability of a completion c_i given the last query term, and the *term to phrase probability* $P(p_{ij}|c_i)$, the probability of phrase p_{ij} extending c_i. In order to promote rare, but distinctive unigrams as completions, $P(c_i|Q_t)$ is the *tfidf* of the completion c_i divided by the sum of the *tfidf*s of all possible completions. The frequencies of bi- and trigrams are likewise normalized:

$$freq_{norm}(\text{order } m \text{ n-gram } p) = \frac{freq(p)}{log(avgFreq(m))} \tag{3}$$

This is to ensure that n-grams with larger n have the same chances of being chosen as n-grams with smaller n despite their naturally lower raw frequencies. $P(p_{ij}|c_i)$ then is the $freq_{norm}$ of the phrase p_{ij} divided by the sum of $freq_{norm}$ of each possible phrasal extension for c_i.

The second part of equation (1) is the *phrase-query correlation* $P(Q_c|p_i)$, the probability of the already completed query terms Q_c preceding Q_t occurring together with candidate phrase p_i:

$$P(Q_c|p_i) = \frac{|D_{Q_c} \cap D_{p_i}|}{|D_{p_i}|} \tag{4}$$

Here, D_x is the set of documents in which an expression x occurs. D_{p_i} is simply the intersection of the document sets of the words in p_i.

After $P(p_i|Q)$ has been computed for all candidate phrases, the ten phrases with the highest probability are returned to the user as query suggestions.

3 Implementation

The components of my query suggestion system can be divided into two main parts, as illustrated by Figure 1: A preprocessing module and the actual query suggestion module. As the name suggests, the preprocessing module does only need to run once. It extracts the necessary n-gram data from the corpus and stores it space efficiently for the query module to use. Once this data has been

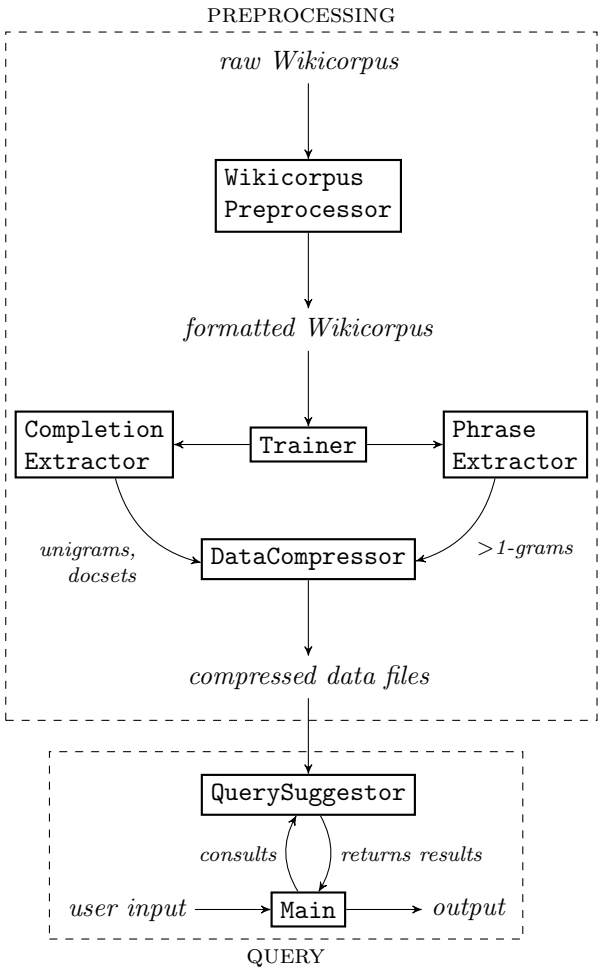

Figure 1: The different components of my program and how they interact.

generated, the preprocessing module does not need to be run again, unless the corpus has been updated. The query suggestion module is the part of the program the user interacts with and which generates the query suggestions.

In the following sections, I explain the structure of the two modules in more detail. First, I discuss my implementation of the query suggestor itself. Then, knowing what kind of data is needed, I turn to the preprocessing module to show how to extract and store that data.

3.1 Query suggestor

My query suggestion system is largely a reimplementation of BMM2011, which I have outlined above. However, I have implemented some small changes to speed up the retrieval process.

The user interface with the system is the `Main` class. This class receives queries

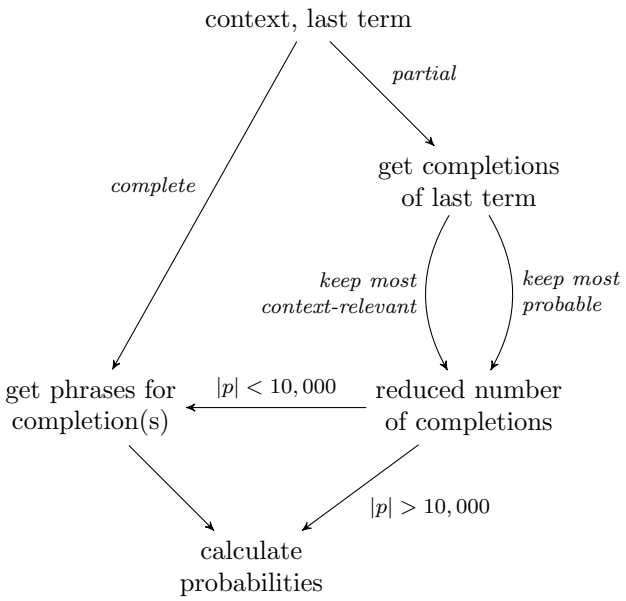

Figure 2: The flow of the `QuerySuggestor` depending on the last query term and the size of the candidate set.

from the user and hands parts of them to the `QuerySuggestor`, which gathers and returns the most probable suggestions. Its workflow is illustrated in Figure 2. The user's input is split into three parts: The *last query term*, which may be incomplete, the *context*, which is the finished term preceding the last term, and the *greater context*, which is everything else preceding the context. Both context and greater context may be empty. For reasons of simplification, only the last term and the context are used by the `QuerySuggestor` in its calculations; the greater context will only be prepended to the results in the end.

With unfinished queries, there are generally two possible scenarios: Either the user is still entering the last query term or they have just finished the last query term and are going to write a new term next. Just as Bhatia, Majumdar, and Mitra 2011, I assume that the last query word is complete if it is followed by a whitespace and that it is partial otherwise. Depending on this, the `QuerySuggestor` will proceed differently.

For a partial term, it first collects all possible completions. Since the unigrams are stored in a trie structure, these completions can easily be retrieved via a prefix search. Then, unlike in BMM2011, a large number of completions is discarded again. The reason for this is that the system is slower the more completions and thus more phrases it has to process. For a query suggestion system, which is supposed to give the user almost immediate feedback, this can be fatal. Especially when partial term is only one or two characters long, there will be thousands of candidate completions, which may again be contained in thousands of phrases each.

In order to keep the most relevant completions, they are first sorted. If the context is not empty and a known unigram, the completions are sorted according

to their *completion-query-correlation* $P(Q_c|c_i)$, which is basically the same as the *phrase-query-correlation* from equation (4):

$$P(Q_c|c_i) = \frac{|D_{Q_c} \cap D_{c_i}|}{|D_{c_i}|} \tag{5}$$

If the context is empty or unknown to the data, the completions will be sorted according to their *tfidf*s. After sorting, only $\sqrt{|c|}$ of the completions will be kept, with a minimum of 10 and a maximum of 100.

Next, the `QuerySuggestor` checks how many phrases this reduced set of completions can produce. If the number exceeds 10,000, no phrases will be retrieved. Again, this is an innovation to guarantee a reasonable runtime. The probability calculation for phrases is very time-consuming and impossible to finish in an acceptable amount of time for more than 10,000 phrases. Hence, the `QuerySuggestor` will calculate the final probabilities for the completions themselves in this case. The equivalent to equation (1) for completions is composed of the term completion probability and the completion-query correlation:

$$P(c_i|Q) = P(c_i|Q_t) \times P(Q_c|c_i) \tag{6}$$

If the last term of the query was already complete or if the number of phrase candidates for a partial term is smaller than 10,000, all phrases for the completed term(s) are retrieved and their probabilities are calculated as outlined in section 2.

The final results are then the top 10 most probable completions or phrases. In case of a partial last term and phrasal results, the most frequent completion in the returned phrases is prepended to the results to give not only phrasal results.

3.2 Preprocessing

In order to be able to do all the calculations mentioned in sections 2 and 3.1, the following data is needed:

- unigrams with *tfidf*s
- unigrams with the sets of documents their occur in
- bigrams and trigrams with normalized frequencies

The preprocessing module is responsible for extracting this particular information and storing it in a space-efficient format.

The data underlying my system is the English part of the Wikicorpus compiled by Reese et al. 2010. It contains about 600 million tagged and tokenized words from 1,359,911 documents extracted from a 2006 Wikipedia dump. I chose the Wikicorpus because it retains information about where a document begins and ends, which is important for the calculations in the query suggestions module, and because of its reasonable download size (2.3 GB when compressed, 9.4 GB

```
The the DT 0
school school NN 0
's 's POS 0
educational educational JJ 0
philosophy philosophy NN 0
was be VBD 0
influenced influence VBN 0
by by IN 0
that that DT 0
of of IN 0
Outward_Bound outward_bound NNP 0
founder founder NN 0
Kurt_Hahn kurt_hahn NNP 0
. . Fp 0
```

Figure 3: A sentence in Wikicorpus format. The columns contain (from left to right) token, lemma, part of speech and WordNet sence. Only the raw tokens are needed for the query suggestion task.

when unzipped). Also, the tokenized format is convenient for extracting the n-grams.

3.2.1 Reformatting the corpus

However, the raw Wikicorpus contains a lot of information that is unnecessary for n-gram extraction. Hence, the first program in the preprocessing chain, the `WikicorpusPreprocessor`, strips the Wikicorpus files from unneeded information and converts them into a format that can easily be processed by the n-gram extractors.

First of all, only the raw tokens are kept, while lemmas, parts of speech and WordNet senses are discarded. All isolated punctuation, URLs and HTML tags are removed and tokens previously divided by an apostrophe (such as *Noun + 's* or contractions like *don + 't*) are contracted. The remaining words are converted to lower case and printed on one line per sentence, divided by single whitespaces. The underscores used to contract multi-word proper names as well as in-word dashes are also replaced by whitespaces. Document boundaries are marked by a <*newdoc*> tag.

Thus, the Wikicorpus sentence in Figure 3 is converted to:

```
the schools educational philosophy was influenced by that of
    outward bound founder kurt hahn
```

After formatting, the Wikicorpus has been reduced to a size of 2.89 GB.

3.2.2 Extracting the n-grams

The preprocessed corpus data is then fed to the `Trainer`, which oversees the generation of n-grams and document sets. Depending on the range of n the

user requests from the `Trainer`, it hands the corpus data to a number of `PhraseExtractor`s and, if unigrams are requested, a `CompletionExtractor`. For the system presented in this report, I generated uni-, bi- and trigrams, but `Trainer` can be used to extract any length of n.

The `PhraseExtractor` is used for extracting n-grams with $n > 1$. It iterates through each sentence and concatenates the words to an n-gram until it has seen n words. However, like BMM2011, it ignores medial stop words. This is because while stop words should not be directly counted into the n-gram due to their high frequency and thus uninformativeness, we cannot completely discard them because they do contribute a meaning to a phrase (cf. 'president of usa' vs. 'president in usa'). Hence, stop words are stored in the n-gram when they are not in initial or final position, but never counted. Thus, both 'president of usa' and 'president in usa' are bigrams even though they contain three words. I also apply the same treatment to numbers, since they are not really words by themselves, but definitely contribute to a phrases meaning. The stopwords used in my system are taken from the default English list of Ranks NL (Doyle 2017).

Because the corpus is so large and the collection of n-grams takes up a lot of space, the `PhraseExtractor` will remove n-grams that only occurred once so far before it runs out of memory. Thus, the frequencies recorded may not be exact. Still, these cleanups are so rare that they will not affect the results to a large degree.

Unigrams are not extracted by the `PhraseExtractor`, but by the `CompletionExtractor`. The reason for this is that they can be extracted more efficiently, since a word that is neither a stop word nor a number (and has more than one character, a special restriction for unigrams) can be stored directly and does not have to be concatenated with other words. Also, the document sets can conveniently be built while extracting the unigrams. Finally, unigrams are stored with their *tfidf*s and not with their normalized frequencies like longer n-grams.

3.2.3 Efficient storage

When the `Trainer` has piped all of the corpus material to the n-gram extractors, it hands them over to the `DataCompressor`. This program is responsible for storing the data extracted by `PhraseExtractor` and `CompletionExtractor` on disk in a memory-efficient way.

One method to achieve this, which will especially save a lot of space on later usage by the query suggestion module, is converting each word into a unique integer ID. The `DataCompressor` thus keeps a map from strings to integers, which it updates with every n-gram it is fed. When writing the n-gram and document set data to files, it will only output these integer IDs. The word-ID mappings can be printed to a separate file in the end for later decryption.

Since all the data now only consists of integers and floating point numbers, they can be stored as bytecode instead of plain text. An n-gram is output as a series of 4-byte integers, each integer being an identifier for one word in the n-gram.

```
q u e r y \t 1 4 9 4 6 . 6 8 3 2 5 0 5 0 1 4 9 4 \n
1 1 1 1 1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  1  = 25 bytes

181618 -1 14946.683...
   4    4        8        = 16 bytes

q u e r y \s r e s u l t s \t 6 . 8 7 9 2 9 4 2 3 4 2 3 2 6 6 7 \n
1 1 1 1 1  1  1 1 1 1 1 1 1 1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  1  = 32 bytes

181618 164992 -1 6.879...
   4        4     4      8      = 20 bytes
```

Figure 4: Bytes needed to store a unigram and a bigram as plain text and as primitive data types. The space savings are significant even for short English strings where each character only needs a single byte in UTF-8. The size of the compressed version is fixed for each n-gram of size n, while the size of the plain version will grow if the strings are longer and contain more special characters.

A negative integer (-1) signals the end of the n-gram and is followed by the normalized frequency (or *tfidf* in case of unigrams) as an 8-byte double. This again saves a large amount of disk space, as illustrated in Figure 4. While the size of plain text files grows with the number of characters in a word (and the number of digits needed to represent a number, since these are also characters then), the number of bytes needed to store an integer or double remains constant independent of the size of that number. The size of the resulting files is thus only dependent on n, i.e. the number of words stored in an n-gram (plus the average number of stop words included in addition).

Since the corpus I used is quite large, efficient data storage is important. My document set file initially occupied about 1.3 GB of space, which I could reduce to about 730 MB, which is a little bit more than half of the original size. It could be reduced even further, e.g. by storing the integers in 3 bytes instead of 4.

To sum it up, the preprocessing module takes the already tokenized Wikicorpus files as input and extracts n-grams of length 1, 2 and 3 as well as mappings from unigrams to the IDs of documents they occur in and a mapping from all words occurring in all n-grams to unique integer IDs. The resulting data files are the basis for the actual query suggestion module.

4 Evaluation

In the extracted data submitted with this program, I have discarded all n-grams with a raw frequency of 5 or less, which yields 683,780 unigrams, 4,169,458 bigrams and 1,138,122 trigrams. This is far more than what Bhatia, Majumdar, and Mitra 2011 tested their system on.

As noted initially, the underlying corpus, Wikipedia, is far more general and covers a broader range of topics than the two corpora used by Bhatia, Majumdar, and Mitra, a collection of Financial Times news articles and discussion threads

from an Ubuntu forum. This is also reflected in the results: Table 1 contrasts some examples of the suggestion generated by Bhatia, Majumdar, and Mitra 2011 and my system. Querying `mount` using the Ubuntu forum data yields results exclusively dealing with mounting a file system, while my Wikipedia-based implementation returns mostly phrases about mountains. Similarly, the Financial Times-based system only returns political phrases when given the query `falkland`, while my system also retrieves geographical information such as 'west falkland' and 'east falkland' or 'malvinas', the Spanish name for the islands. With context, the results become more similar.

Thus, my non-specific system is indeed able to come up with meaningful queries despite the variation in its data. However, it is unable to serve specific information needs without context specification. In order to get suggestions for `mount` similar to those proposed by BMM2011, one needs to query `filesystem mount`.

Unfortunately, the system performs poorly on very short partial queries, even when given context. Consider the results for `screen r`:

```
> screen restruct
> screen rectaflex series
> screen runefaust army
> screen harry roedecker
> screen power rensa
> screen rensa sibari
> screen genseijuu riseross
> screen von rheingarten
> screen rudolf rassendyll
> screen restruct destruct o lux
```

None of the phrases returned is a sensible extension for `screen r`, even though both phrases and completions are filtered for correlation with context. It seems that the frequencies of the phrases still have a large impact on their position in the results or that co-occurrence within a document is not a sufficient criterion for phrase-query-correlation. I experimented with promoting completions that co-occur with the context in n-grams, but this pushed other uninformative phrases to the top that frequently co-occurred with any word, so a larger stop word list might also be helpful.

5 Conclusion

Basing a query suggestion system on n-gram data in the absence of sufficiently large query logs is a good method for achieving reasonable suggestions. However, the quality of the results is very dependent on the nature of the underlying corpus, and is higher if both n-gram data and field of application are centered around the same specific topic. It could be beneficent to use the very data one intends to query or a similar corpus for extracting the n-grams to ensure that the system makes sensible suggestions.

In general, the results could also be improved by giving more weight to the candidates' correlation with the rest of the query to avoid suggestions that

obviously have no thematic connection to what the user has typed before. Also, my implementation currently only evaluates phrases and completions against the final context term. It would be interesting to see whether it can generate more meaningful suggestions when given the complete context.

Another idea for finding phrases related to the context and completed last query term would be to make use of a clustering method to retrieve phrases or single words in the vicinity of the query terms. This could also be useful for getting a broad range of different suggestions, as opposed to variations of the same term, by extracting a selection of candidates that is evenly distributed in the cluster space. Such an approach could solve the non-specificity problem with the query `mount` by generating diverse suggestions such as `mountain`, `mount horse` and `mount filesystem`, so that a narrowing context does not have to be provided by the user on their own but can be selected from the query suggestions.

References

Bhatia, Sumit, Debapriyo Majumdar, and Prasenjit Mitra (2011). "Query suggestions in the absence of query logs". In: *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. ACM, pp. 795–804.

Doyle, Damian (2017). *Stopword Lists*. URL: `http://www.ranks.nl/stopwords`.

Reese, Samuel et al. (2010). "Wikicorpus: A word-sense disambiguated multilingual wikipedia corpus". In: *Proceedings of 7th Language Resources and Evaluation Conference (LREC'10), La Valleta, Malta*. URL: `http://www.cs.upc.edu/~nlp/wikicorpus/`.

BMM2011	My implementation
query: `mount`	
mount	mountain
unable to mount	mountain range
mount point type	rocky mountains
sudo mount	rocky mountain
able to mount	mount vernon
mountpoint	mountain ranges
try to mount	mounted police
mount the drive	mount pleasant
mount the partition	mountainous terrain
file system mount	mountain view
	3375 ms.
query: `falkland`	
falklands	falkland
falklands war	falkland islands
falkland islands	falklands war
falklands conflict	east falkland
1982 falklands	falkland islands dependencies
1982 falklands conflict	viscount falkland
falkland islands government	west falkland
falklands war in 1982	falklands malvinas
1982 falklands war	battle of the falkland islands
invasion of the falklands	falklands conflict
	16 ms.
query: `screen resoluti`	
screen resolution	screen resolution
screen change the resolution	screen high resolution
screen native resolution	screen higher resolution
screen set the resolution	screen low resolution
preferences screen resolution	screen resolution graphics
screen correct resolution	screen display resolution
screen resolution and refresh	screen resolutions
screen low resolution	screen high resolution graphics
screen monitor resolution	screen resolution images
screen comparing resolution	screen vertical resolution
	203 ms.
query: `encryption equip`	
encryption equipment	encryption equipment
encryption digital equipment	encryption digital equipment corporation
encryption office equipment	encryption digital equipment
encryption electronic equipment	encryption communications equipment
encryption telephone equipment	encryption electronic equipment
encryption equipment and services	encryption cryptographic equipment
encryption video equipment	encryption standard equipment
encryption medical equipment	encryption audio equipment
encryption transmission equipment	encryption fully equipped
encryption original equipment	encryption radio equipment
	875 ms.

Table 1: Query suggestions generated by BMM2011 and my system, and the response times of my system.