Query Expansion using Word2Vec

Course project CS582

Piervincenzo Ventrella
Computer Science
University of Illinois in Chicago
Chicago, IL
pventr2@uic.edu

ABSTRACT

This is the report for the final project of the CS582 course in the University of Illinois in Chicago. For the final project we choose to implement a search engine. The search engine is based on a tf-idf scheme to which, next, a query expansion algorithm is added, in order to improve the quality of the retrieved documents. In this paper I investigate the applicability of Word Embedding in the context of information Retrieval. The purpose of the project is to observe the performance of the Search Engine when the query of the user is expanded with the most similar terms according to Word2Vec models. Word Embeddings makes use of Neural Networks that need to be trained on a collection of words appearing in different sentences. For this reason, it could be interesting to compare the results that will be obtained when the training is performed globally (using a pre-built model on a dataset provided by Google) and locally (using as dataset the sentences contained in the documents that the System will crawl and index). Word Embedding and Neural Networks have a lot of applicability in the context of Information Retrieval and Natural Language Processing, here we investigate only the benefits in using them for query expansion at retrieval time. Further works and different applications are discussed later.

KEYWORDS

Search Engine, Information Retrieval, tf-idf, query expansion, Word2Vec, UIC

1 Introduction

Searching for pages in World Wide Web is perhaps the most widely used IR application.
Given a corpus of textual natural language documents and a user query in the form of textual string, the task of a Search Engine is to provide to the user a ranked set of documents that are relevant to the query.
The project is structured among the 3 main parts a Search Engine is composed of:

- **web Crawler**, the component used to crawl the web pages that will compose the documents' collection;
  In this scenario we decide to stay inside the university d omain and those the Crawler starts visiting the https://www.cs.uic.edu URL and performs a Web traversal using a breadth-first strategy following the links present in the pages and being careful to remain inside the domain .
- **Inverted index**, the component used to index those documents and allow to efficiently retrieve the relevant document at query-time.
- **Retrieve component**, the component that takes the user query and ranks the indexed documents of the collection according with some criteria such as tf-idf score. This component also offers to the user a Graphical User Interface to facilitate the interaction with the System and the results visualization.

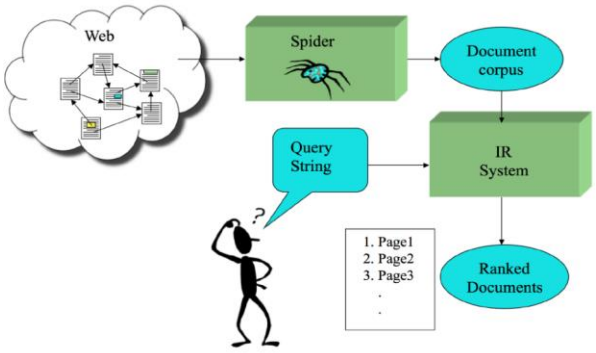

Figure 1: General Structure of an Search Engine. The spider is the agent in charge to crawl the pages that will form the document corpus. The documents are then indexed. At query time the documents are ranked with respect to the given query.

In this paper we discuss how the different Word2Vec models provide similar words. We also provide a manual evaluation of 5 queries comparing the results when query expansion is performed or not. Finally, we discuss the results, other possible approaches and future work.

2 Crawler

As mentioned before, the Web Crawler must visit and save the text content of the visited web pages being also careful to remain inside the UIC domain.
To do that, in this phase, we define a spider that starts visiting https://www.cs.uic.edu URL. While visiting the page, the spider parses the content and extract both, other contained links (usually of the form) and the text

content. Web pages can be html files, pdf or other formats so they need to be parsed differently. We choose to parse only html and pdf documents and we keep in the collection only the pages with at least three words (this because sometimes different formats produce an empty parsed text). The spider, in traversing the web, adopts a Breath-First s strategy. To efficiently crawl the web, we keep a list of already visited links and we avoid parsing twice the same page. In doing this we have also to normalize the links: this means for example remove the "/" at the end of the URL or considering as equals the links that are the same except for starting with "http:" or "https:". Finally, we remove from the link also the suffixes composed by "#something" that just remand to different sections of the same page.
In saving the document, together with the text content we include also the link itself that usually contains descriptive words of the document.
We crawl 3200 pages.

```
dictionary = spider("https://www.cs.uic.edu", 3200)
```

```
0 Visiting: https://www.cs.uic.edu
text/html; charset=utf-8
done 0
1 Visiting: http://engineering.uic.edu
```

Figure 2: Example of page Crawling. The spider starts from the given URL and performs a Web traversal using a breadth-first strategy.

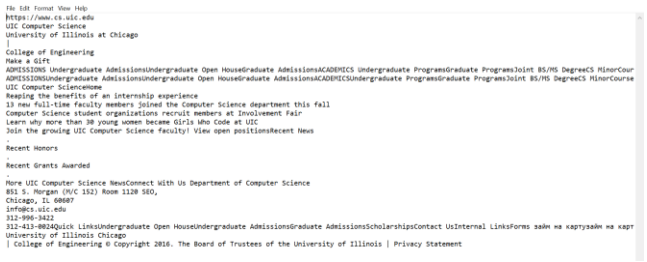

Figure 3: Example of saved text content. The text content of the page is saved into file. The link itself is also included.

3 Inverted Index

The main purpose of the inverted index is to allow at query-time to efficiently retrieve and rank the relevant document with respect to the given query. To do that the Inverted Index consists of a dictionary with words in the collection as entry and the list of containing documents as value. This allow to retrieve only the documents that contain at least one word of the query, reducing the need of computation in ranking the documents.

In building that dictionary we process each document in the collection and transform the text in a list of words. When we extract the words, we perform:

- Tokenization: we extract from the text only words composed by alphabetical characters, eventually separated with "-" and we put those words in lower case;
- Stemming: for that purpose, we use the PorterStemmer library available in python.
- Stop Words removal: we do not include in the Inverted Index words that are contained in a list of common stop words.

After building the Inverted Index we also compute the weighted norm of each document that will be useful during the ranking phase.

4 Retrieval

To perform this task, separately from the others, we load from file all the needed documents such as the Inverted Index, the Document weighted Norm dictionary, the dictionary of crawled URLs. We load also the two Word2Vec models (the pre-trained model on GoogleNews and the one trained on our collection). Then we launch a Graphical User Interface to interacts with the user and asks for the query input.

There are two main actions that this component performs: Query Expansion and Ranking of the documents.

4.1 Query Expansion

When the user inserts the query, he has three options: search the results using one of the two Word2Vec models that performs query expansion or searching without performing query expansion.

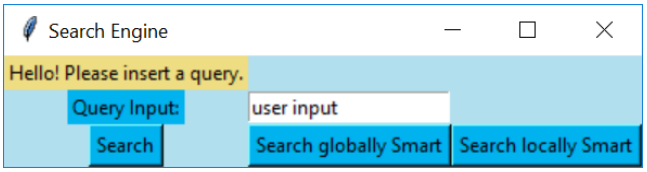

Figure 4: GUI. The user can choose to use one of the two query expansion methods or to not perform query expansion.

If the user chooses "Search globally Smart" the query will be expanded using the pre-trained model provided by Google. The Systems takes each single word in the query and look for the most similar words that are then added to the original query. To be precise only the words that have a similarity value greater than 0.5 are added to the query. This is done to not include irrelevant words that will decrease the precision of the Search Engine. Symmetrically, if the user clicks on "Search locally Smart" the query is expanded using the model trained on the collection of documents. The comparison of the two models is discussed in the next sections.

After the expansion (or without performing it) the query is processed with the same procedure of the documents in the Inverted Index phase (tokenization, stemming, Stop Words

removal, …). Finally, only the documents that contains at least one query's word are ranked**.**

4.2 Ranking

The ranking score is computed according with the tf-idf scheme. The System computes the Cosine similarity between the query and each of the retrieved document. Other possible score functions are proposed later.

$$CosSim(d_j, q) = \frac{\langle d_j, q \rangle}{\|d_j\| \cdot \|q\|} = \frac{\sum_{i=1}^{t} w_{ij} w_{iq}}{\sqrt{\sum_{i=1}^{t} w_{ij}^2 \cdot \sum_{i=1}^{t} w_{iq}^2}}$$

Figure 5: Cosine Similarity. Compute the inner product normalized by vector length. The weights are computed multiplying the term frequencies of the word in the document by its Inverse document frequencies.

The Retrieval Component incrementally compute the score of each retrieved document. Finally, the scores are normalized by the weighted norm of the corresponding document.

Note: in considering the words coming from the query expansion we multiply the tf-idf score of those words by their similarity value. This value is always less than 1 and this is done to not give to those words the same weight of the original words in the query. After the ranking is complete, another GUI is showed to the user to let him visualize the URLs.

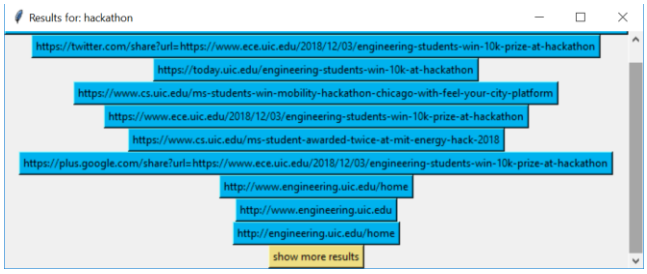

Figure 6: Results for query "hackathon". The first 10 results of the input query are displayed to the user. Eventually the user can choose to see more results.

5 Globally vs Locally most similar words

During the Development of the system we compare also which kind of words are retrieved as more similar depending on which Word2Vec model we use.

The so-called global model is trained on the GoogleNews dataset and it includes 3 million of words. Here an example:

```
similar("computer")
```

```
[('computers', 0.7979379892349243),
 ('laptop', 0.6640493273735046),
 ('laptop_computer', 0.6548868417739868),
 ('Computer', 0.6473334431648254),
 ('com_puter', 0.6082079410552979),
 ('technician_Leonard_Luchko', 0.5662748217582703),
 ('mainframes_minicomputers', 0.5617721080780029),
 ('laptop_computers', 0.5585449934005737),
 ('PC', 0.5539618134498596),
 ('maker_Dell_DELL.O', 0.5519254207611084)]
```

Figure 7: Most similar words to "computer" with global model. The global model retrieves the most similar words to the given word, together with their similarity values.

On the other hand, the local model is trained on the document's corpus. To do so we built a vocabulary containing the text of all documents. Using the nltk library available in python we preprocess the text and extract the sentences used as train set. The model contains 4.7 million of words. To train the model we tune the hyperparameters such as min_count (the minimal number of occurrences in the vocabulary in order to consider the word for the training, we choose a value of 2) and window_size (the max_distance to consider 2 words as adjacent, we choose a size of 3). Here an example:

```
sim_words("computer")
```

```
[('digital', 0.991154134273529),
 ('services', 0.9911483526229858),
 ('media', 0.9910385608673096),
 ('student', 0.9910367727279663),
 ('office', 0.9907900094985962),
 ('alternate', 0.9900556802749634),
 ('drc', 0.9900108575820923),
 ('accessibility', 0.9899654984474182),
 ('complete', 0.9898406267166138),
 ('fax', 0.9898104667663574)]
```

Figure 8: Most similar words to "computer" with local model. The local model retrieves the most similar words to the given word, together with their similarity values.

Another example is provided:

```
similar("student")
```

```
[('students', 0.7294867038726807),
 ('Student', 0.6706663370132446),
 ('teacher', 0.6301366090774536),
 ('stu_dent', 0.6240992546081543),
 ('faculty', 0.6087332963943481),
 ('school', 0.6055628061294556),
 ('undergraduate', 0.6020306348800659),
 ('university', 0.6005399823188782),
 ('undergraduates', 0.5755698680877686),
 ('semester', 0.573759913444519)]
```

```
sim_words("student")
```

```
[('digital', 0.9986699819564819),
 ('accessibility', 0.9978376626968384),
 ('services', 0.9978151321411133),
 ('complete', 0.9977531433105469),
 ('campus', 0.9976484775543213),
 ('uic', 0.9975920915603638),
 ('search', 0.997422993183136),
 ('form', 0.9974071383476257),
 ('drc', 0.9972301125526428),
 ('exam', 0.9972104430198669)]
```

Figure 9: Most similar words to "student" with global (above) and local (below) model.

From the 2 model we notice first that the local similarity values are higher. For this reason, we multiply the weight of the words expanded with the local model by an additional factor of 0.9, always in order to weigh less the words that are not contained in the original query. We can notice also that the global model tends to provide as similar the original word but slightly modified (i.e. the plural form or the same word starting with capital letter). When these words are processed and stemmed, they will be identical to the original one and will not add new relevant terms to the query but will only increase the weight of the single word (the word will appear more than once in the expanded query). Probably this is not what we want.

Finally, the similarity values are proportional to the observed frequency of the two words appearing in the vocabulary as adjacent. For further explanation how Word2Vec works check the References section.

6 Evaluation

Here we provide some manual evaluation comparing the retrieved document for the different searching options. Below a comparison for the query "hackathon".

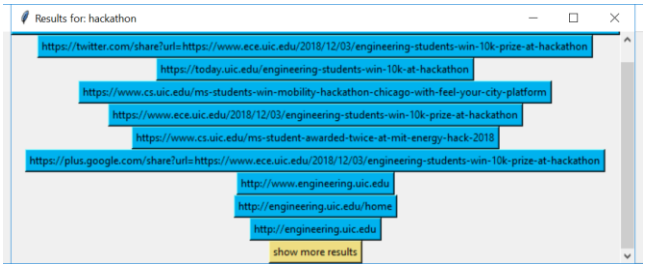

Figure 10: Results for query "hackathon" without query expansion. How we can notice from the top 10 results almost all the URLs are inherent to the word "hackathon" except maybe for the last 3 results that bring the user to different main pages of the uic domain. This can be explained by the fact that, when the crawling was performed, in the news section of those pages there were posted articles of the hackathon awards. Finally, the first 2 results send to posts of these hackathons published on twitter or Facebook. Even if these posts are still related to the query, maybe these 2 results could be irrelevant to the query. Overall, we can be satisfied of the proposed results. Without performing expansion, 51 documents match the query.

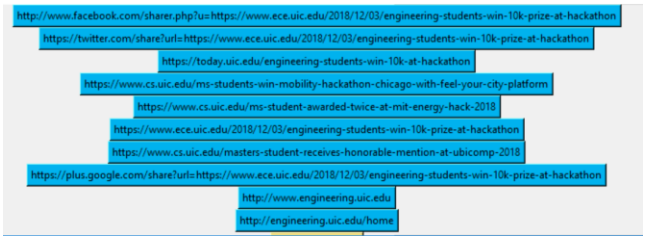

Figure 11: Results for query "hackathon" with global query expansion. How we can notice, the results are pretty similar to the previous case where no expansion is performed. We just notice that one of the last irrelevant URLs is replaced with "masters-student-receives-honorable-mention-at-ubicomp-2018" that is more relevant to the query. In this case 56 documents are matched and the expanded query before the preprocess is:

```
"hackathon Hackathon iPhoneDevCamp hackathons RHoK hack_thon barca
mp unconference".
```

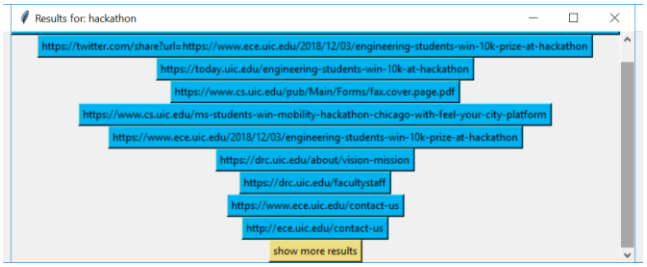

Figure 11: Results for query "hackathon" with local query expansion. Here we can still notice the presence of relevant documents retrieved also in the cases above. We also notice the presence of more irrelevant URLs in the last positions. The expanded query is:

```
"hackathon fax arrow proctoring map admissions health vision home
st disabilities"
```

The matched documents are 2804, much more of the previous cases. Below a table containing other manual evaluated queries.

Query	No Expansion	Global Expansion	Local Expansion
Artificial Intelligence	191 documents almost all relevant	197 documents almost all relevant	191 documents almost all relevant
Cornelia Caragea	7 documents 4 relevant	10 documents 5 relevant	8 documents 4 relevant
Courses catalog	1212 documents 10 relevant	1510 documents 9 relevant	3012 documents 4 relevant
conference	314 documents 4 relevant	314 documents 4 relevant	314 documents 4 relevant

Figure 12: manual evaluation for 4 additional queries. In the table are showed the number of matching documents and how many of the top 10 are relevant.

7 Discussion of results

From the results we can conclude that both query expansions methods add new terms in the query and allow the System to retrieve more documents possibly increasing the Recall values.

Performing query expansion, there is the risk to add terms that are not relevant to the query and this causes to retrieve more irrelevant documents lowering the Precision of the Search Engine. This is true specially in the case where we use the local model. Looking at Fig.11 we notice that the new added terms does not seem really correlated with the word “hackathon”. A possible motivation is that Word2Vec looks for the cooccurrence of the terms in the text. In this way, words that occurs often could have more probabilities of being similar to the original word. Because of that, there are more probability to expand the query with common terms that are included in more documents. More matching documents means more computation, and this forces the user to wait a couple of instants before being able to visualize the results.

8 Conclusion

Query expansion using the most similar words retrieved by a Word2Vec model allows to add new terms to the original user input and retrieve documents that are relevant even if they do not contain the exacts words of the query. This helps to overcome the intrinsic problem of BagOfWords models where the documents are represented only by the words they contain, without considering any particular order. For that reason, query expansion can improve the Recall performances of a Search Engine. Sometimes this is done without lowering the Precision values but in other cases, if the expanded terms are not really related to the original terms, the number of irrelevant documents can increase. The Word2Vec model provided by Google slightly increases the number of matching documents without remarkably improving the Recall but also without generating too many false positive. Vice versa, the Word2Vec model trained on the collection allows to retrieve much more documents increasing the Recall of the System but inevitably generating more false positive and decreasing the Precision.

9 Future work

As future work could be interesting to explore other applications of Word2Vec and not only using them to retrieve similar words. We could use Word Embeddings to represent the Documents and the query itself: rather than using a BagOfWord model we can represent the document in 300 fixed dimensions (as done with single words in Word Embedding) where each dimension value is computed as the contained words’ average value for that dimension. This could allow to express the semantical meaning of the document and the ranking score will not be based only on the matching words between the document and the query. Moreover, this will remarkably reduce the number of features and will speed up the computation currently based on the Inverted Index. The Cosine similarity will be computed on this new dimensions.

If we decide to keep the BagOfWord model as done in the project we could also try to incorporate a Page Rank score together with the tf-idf score. This can help in the scenario where an irrelevant document is retrieved only because the query is expanded with not related words. Weighting the score of a document also with respect to his PageRank score (that is query independent) could reduce the problem.

REFERENCES

[1] https://becominghuman.ai/how-does-word2vecs-skip-gram-work-f92e0525def4 intuitively understand how WordToVecs works.
[2] https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf in this book is provided a complete explanation of WordEmbeddings.
[3] Some theory concept and Images are taken from the slides of the CS582 course, Information Retrieval.