
## CS 510 Final Report

# Ranking Academic Papers Using Relevance Judgements

Keval Morabia morabia2@illinois.edu Shubham Singhal ss77@illinois.edu

Malcolm Tivelius mct6@illinois.edu Tara Vijaykumar tgv2@illinois.edu

**GitHub Repo**: https://github.com/kevalmorabia97/ACL-Search-Engine-CS510

**Search Engine**: http://bedbb9ef.ngrok.io

**Demo Video**: Google Drive Link

NOTE: We were having some issues with hosting the search engine on web.illinois.edu so we hosted it on the EWS machine using ngrok tool. We are not sure if those machines reset everyday or not. So there is a possibility that the link doesn't open in which case we request you to email any of us and we will rerun the search engine so that you can evaluate it.

# Table of Contents

# Introduction

A literature search is a systematic, thorough search of all types of literature (research publications, books, peer reviewed articles) in a topic. A literature search is the most essential part for every researcher to know about the recent trends in a research area and to learn about the research methodologies. Literature searches are commonly performed by scholars as part of research work and/or postgraduate study.

A search engine is a software system that searches the world wide web in a systematic way to retrieve particular information in a search query. It outputs a list of results (Search Engine Result Pages) ordered in decreasing order of relevance. The results may be a mix of links to web pages, images, videos, infographics, articles, research papers, and other types of files. Search engines maintain real-time information by running an algorithm on a web crawler.

A search Engine acts as a filter for the large quantity of information available on the Internet. They allow users to quickly and easily find information that is of genuine interest or value to them. There is a lot of filtering to do - in 2004 the number of pages in Google’s index exceeded 8 billion. With this enormous amount of data, using the internet would be impossible without search engines.

## Search Engine Features

To make our search engine application stand out we have improved upon a standard search engine with four new features. These are the following:

1. A module to search for papers related to a paper of interest to the user
2. A module to mark the retrieved list of papers as relevant or non-relevant this is to be collected from the user and is to be used to learn relevance judgements and improve the algorithm
3. A module to include longer and more generic queries, such as the idea of the user's project
4. Include datasets such as academic paper datasets from Phase 2 (search engine competition), in addition to the ACL Anthology UIUC Corpus.

**The first feature** allows the user to select a research paper from the list of results of a user input search query. Our application will then show the user a list of relevant papers to the one selected. All the documents shown contain at least one of the query words and in the results shown, all query words are **bold** marked.

**The second feature** allows users to improve our search engine by specifying whether a document in the results was relevant or not. The relevance score for each document is used along with the score our search engine gives. From all the results returned by our search engine, "Relevant" marked documents are shown first and then the other documents according to their score. If a user clicks on a document in the results it also marks the document as relevant. However, a user can only mark the document as relevant once per session, so if they mark it as relevant and click on the link it will only give +1 in score. Also, if the user clicks "Relevant" on a document and then "Not relevant" it will neither be marked "Relevant" nor "Not relevant".

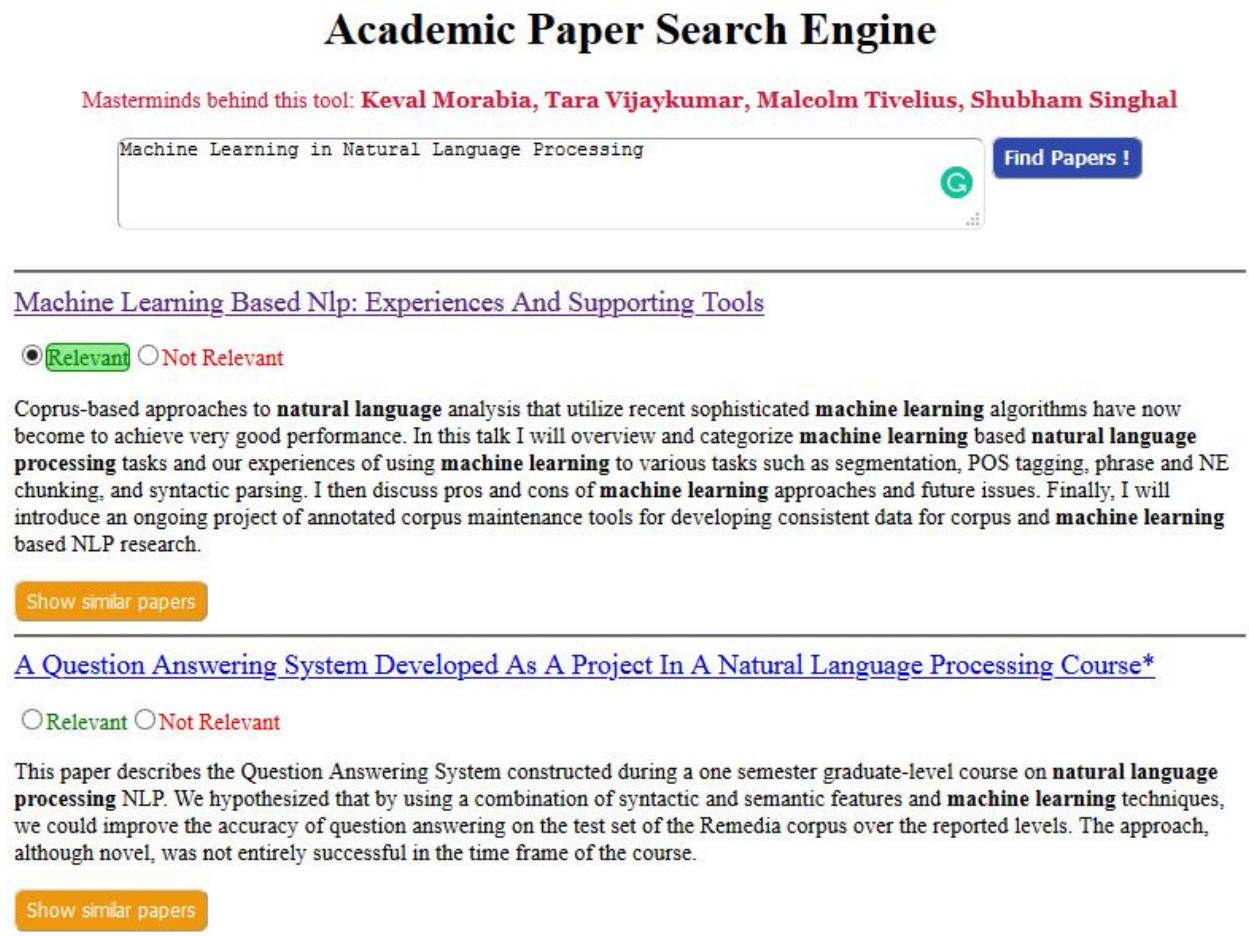


Image 1: Search Engine User Interface showing results for a query. All query words are marked **bold** in the paper abstract. Note that for the first result, "Relevant" got marked automatically since the user clicked on the link. A user can also explicitly mark any document as "Relevant" or "Not Relevant" for a query.

**The third feature** we implemented in our search application is allowing users to specify a long query, defined as having more than 10 words after preprocessing the query. The idea is that the user, instead of searching for a paper by its title, should specify a general idea and describe it in more detail to allow us to find a relevant paper that presents the same ideas.

An example of such a query could be:

> *"The utility of SNAP for various paradigms of natural language processing and what is the performance on several important applications such as the memory based parsing and translation, classification based parsing"*

Which is reduced to:

> *"performance classification important snap based processing various natural several applications paradigms parsing language utility translation memory"*

The keyworded version of this only contains ~50% of the words from the original sentence but still includes most of the important query words. The fact that the query words are shuffled doesn't matter for the final results as we don't consider the ordering of the words when ranking documents.

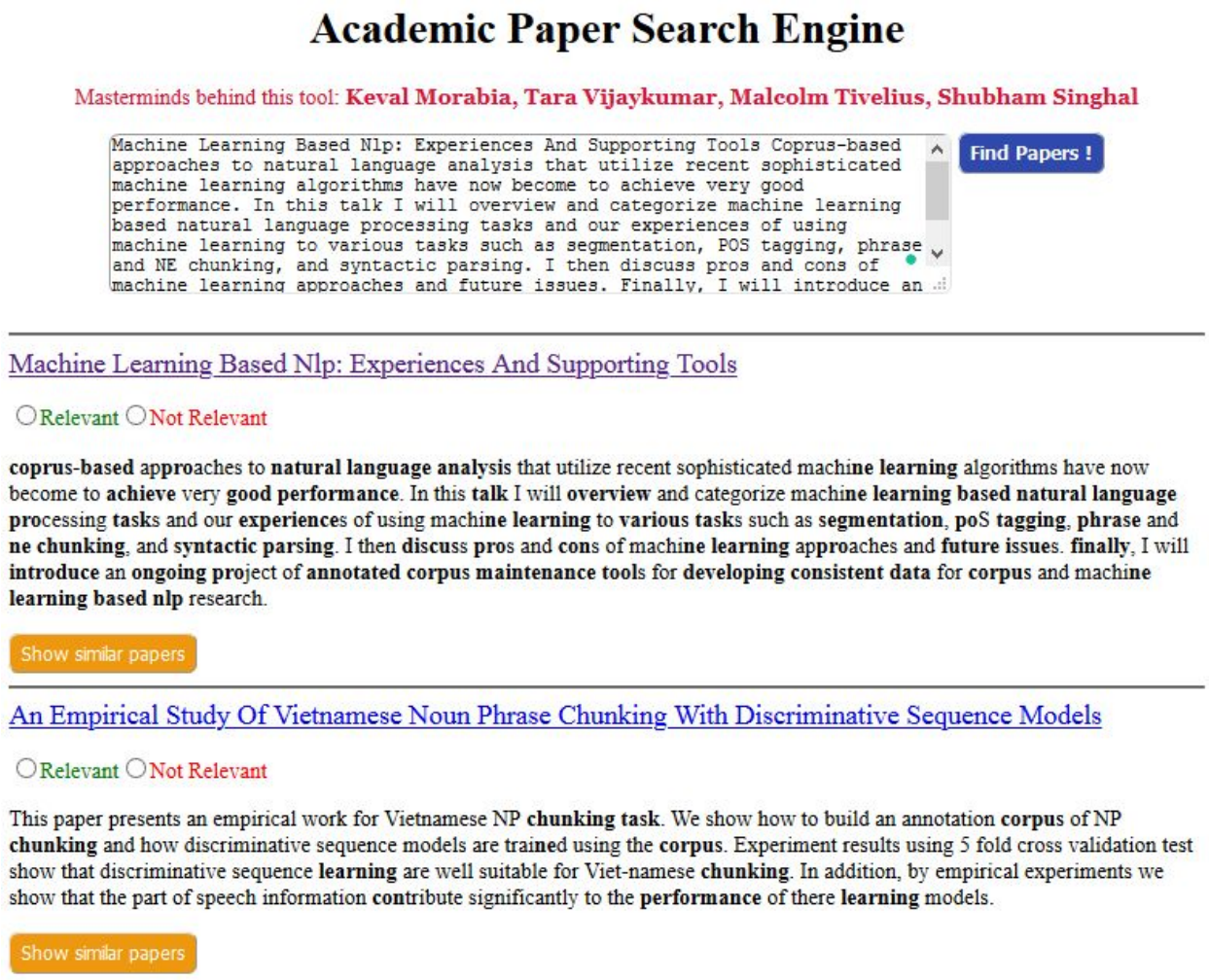


Image 2: Similar paper search for the first result in Image 1. In this case, the title and abstract of the selected paper are used for querying the search engine. In this case, the long query search feature is triggered.

**The fourth feature** is that we include several datasets under the same search engine. They are ACL Anthology UIUC Corpus and Academic Dataset from Search Engine Competition. The former contains a corpus of around 40,000 documents and latter around 8,000 documents. We envisaged to have couple other datasets which are generated by students in the class as part of their object but we didn't have any information about how to obtain them. Nevertheless, we used these datasets in tandem to make our query results richer.

# Algorithms and Reasoning of the Choices

## Preprocessing

During pre-processing, we explored various natural language processing techniques to see which yields the best results for our tasks. We removed stopwords in the document using the English stopwords provided in **Python's NLTK package**. We also used the **WordNet Lemmatizer** from the same package to perform lemmatization. A similar procedure is followed when preprocessing search queries.

Pseudocode for preprocessing documents:

```
// create an empty dataframe to store the results
for each document:
      extract title, abstract, introduction and link
      for title, abstract and introduction:
            tokenize
            remove stopwords
            lemmatize
            convert to lowercase
      append document and tokenized document to dataframe
return dataframe
```

Due to the large amount of documents, we stored 'corpus' and 'tokenized corpus' as Python's pickle objects to improve storage requirements.

Apart from NLTK, we also experimented with the Spacy library for tokenization and lemmatization but this increased the complexity and processing time. Hence we used the NLTK library for faster results.

## Long-Query Algorithm

The way we approach this problem is that we take the long query as input from the user. The first thing we do with the query is to extract keyword phrases from it. During this process we also remove stopwords and punctuation. The phrases are selected using the **NLTK RAKE library**. Furthermore, we have chosen to implement a limit of four words for each of the extracted phrases, this is done to limit the amount of keywords extracted in total. Another way of doing this could be not to limit the RAKE algorithm at all, but then we run the risk of not reducing the input query enough which could affect the search results poorly. The RAKE library also ranks the phrases extracted in order of importance. We have however elected to utilize all phrases and disregarded the ranking in order to not miss any information provided by the user. This was determined after testing different settings, such as only keeping the top 50% of the phrases and not limiting the word count of the phrases. After extracting all the phrases we combine them into one query and remove all duplicate words.
An outline of the algorithm:

```
query := read_input()
If length query > 10
      rake.extract_keywords_from_text(query)
      phrases := rake.get_ranked_phrases()
      text := join(phrases)  #Keeps all of them
      unique_words := set (text.split(' '))
      new_query := join(unique_words)
```

## Relevance collection Algorithm

The user is provided an option to mark a document relevant or not relevant for the query. A document has 2 types of scores for a query: Search engine score and relevance score. All documents initially have relevance score of zero. If the document is

marked relevant, then its relevance score increases by 1. If it is marked not relevant, then its relevance score decreases by 1. For any subsequent query, all returned documents, are sorted first by relevance score, then by search engine score.

## Ranking algorithm

**Okapi BM25** is a ranking function used by search engines to rank matching documents according to their relevance to a given search query. It is based on the probabilistic retrieval framework developed by Stephen E. Robertson, Karen Spärck Jones, and others.

**BM25+** is an extension that was developed to address one deficiency of the standard BM25 in which the component of term frequency normalization by document length is not properly lower-bounded; as a result of this deficiency, long documents which do match the query term can often be scored unfairly by BM25 as having a similar relevancy to shorter documents that do not contain the query term at all.

We used BM25+ as our ranking algorithm as it is an unsupervised ranking method and doesn't require any relevance judgements. Due to the lack of these relevance scores for query and document pairs, we were unable to use a supervised method. After collecting sufficient relevance judgements from users, we will be able to run supervised algorithms that make use of relevance.

**Pseudocode for BM25+:**

```
initialize doc_scores
      for word in search_query:
            word_freq = counts of word in documents
            doc_score += IDF(word) * (delta + (word_frew * (k1 + 1)) /
                  (k1 * (1 - b + b * doc_len / average_doc_len) + word_freq))
      return doc_scores
```

# Novelties and their importance

In this section, we discuss the novel features in our search engine and the importance of each of them. In our search engine, we implemented the four novel features that are not usually part of a literature search engine. In this section, we discuss each novelty as well as its relevance importance.

## Similar Paper Search

Suppose a user finds a paper useful to his search, and wishes to find more papers similar to this one. In most search engines, the user would not be able to do this - they would have to browse through other results or modify their search query. To make it easier for a user to find similar papers, we implemented a novel feature that solves this issue.

Instead of the user having to think about formulating a new query, they can just select the paper of their interest and be presented with a list of similar papers. The reason we believe this feature is important is because it can be really difficult to find similar research papers to a paper you have a use for. Usually it involves looking at citations, but they don’t always lead you where you want to go. This feature can be used by students and professors alike, trying to either develop applications backed by research, or to do research of their own.

## Long Queries

We aimed to develop a search engine that could be used by both researchers as well as users curious to know more about a certain topic. Compared to academic experts, users generally don’t have much domain-specific knowledge and hence their queries are

more likely to be broad and generic. If we were to search for this entire query, we may not get relevant results, or we may not get results at all. Therefore, we decided to include a feature that would handle this issue.

Another important side of this feature is that it allows more experienced users to write a detailed query that isn't necessarily a search for a specific paper but rather a search for papers that have proven or disproven a certain thing. This makes it possible to return documents with valuable information you can refer to, even though the papers main topic may be about something else. Generally, this is a feature that is not common to traditional search engines and it makes our system more usable.

## Mark the Results Relevant or Not Relevant

A user might just click on documents out of curiosity and not because it was relevant - measuring the clicks may not necessarily be a good measure to determine relevance. To improve the quality of our search engine, we explicitly ask the user to mark a document as relevant or non-relevant. This is especially useful when done by domain experts who would have a better knowledge of what might be relevant to a particular search term.

We collect these relevance judgements and use it to improve document relevance ranking. Adding this feature gradually improves the quality of our search engine's results and is also useful for future users.

## Including Papers from Various Datasets

We wanted to make our system more robust and include more datasets for training our search engine, so that it would be able to provide better content and results to the user.

So we included more literature papers from the Search Engine Competition (Academic Paper Dataset). Also the aim was to be a one stop solution for academic search engine. Thus having more datasets would help this cause. But as said above, we could not obtain the datasets from other students in the class and thus used the two available.

# Challenges and their Solutions

In this section we briefly discuss the major challenges faced and how we solved them.

## Missing data during preprocessing

Some papers did not have titles, abstracts and introductions. Some papers also gave us encoding errors when we tried to preprocess them. We handled this issue by skipping papers that did not have both abstract and introduction. To make up for the loss in training data due to these skipped papers, we added another dataset, the Academic Papers Dataset to make the system more robust.

## How to collect relevance

A user might just click on documents out of curiosity and not because it was relevant - measuring the clicks may not necessarily be a good measure to determine relevance. To overcome this issue, we decided to collect relevance judgements by explicitly asking the user to mark a document as relevant or non-relevant. This is especially useful when it is done by a domain expert.

## How to search for similar papers

Initially, we decided to try searching for similar documents by making the title of the selected document as the search query. However, this was returning relevant results as the title of the paper may not contain the keywords most relevant to the idea of the paper.

To overcome this issue, we decided to formulate the search query using the selected document's abstract. The abstract of a document is usually a summary of a paper

consisting of the main keywords and features and hence using this as the new search query made more sense.

## Deployment issues

We had some issues related to hosting our search engine on web.illinois.edu that was mainly caused due to the following issues:

- Limit on memory storage - unable to store a large amount of data
- Installing the nltk package on the server took up too much space

To overcome these issues, we hosted the search engine locally and then shared the link using ngrok. We plan to later host this on AWS/GCP.

## User Guide

For anyone to run our application, they can clone the GitHub repository mentioned on the first page. After cloning, all zip files in “data” directory have to be unzipped.

All required Python3 packages are mentioned in “requirements.txt” file. Run the following command to install them:

***pip3 install -r requirements.txt***

Now, the search engine is ready to be used! To run the search engine, use the following command:

***python3 run.py***

It will take about 1 minute for the initialization of the search engine after which it can be used at ***http://localhost:5000/*** on the same machine.

# Conclusion

We thus implemented Academic Search Engine with novel features like searching for similar papers, giving users the option to mark the result as relevant or not, giving an option to search for long queries etc. In the process, we faced a lot of challenges as described above and had to make various design choices to best complete our work. In the end, we learned a lot about the information retrieval problem and the scope of improvement in various areas related to it. Of course, there is a lot that can be improved but this project has helped us learn the basics, not only theoretical details but also practical issues in implementing a search engine. We hope to improve our search engine further by implementing various state of art ranking algorithms and benchmarking our performance with current academic search engines. The search engine competition helped us a lot in tinkering with various algorithms and learning about the processing of various datasets. The lessons learned from there also helped in making our project better. Finally, we just wish to say we thoroughly enjoyed working on this project and hope to improve it in the future.