guUCLe

A search engine implementing Learning To Rank for the UCL website

Emmet Cassidy *SN 900138*

Jason Cheung *SN 15081356*

David Kelly *SN 15097132*

Nicholas Read *SN 15084428*

April 2016

Contents

Abstract

The purpose of this project was to build a search engine that indexes and retrieves content from the *ucl.ac.uk* domain. It was decided to restrict the content to text rather than images/video, and to restrict document crawling to a sufficiently large sample of the domain, rather than attempt to crawl all ¿ 1 million urls[1] To evaluate the results, we were required to generate a suitable set of test query topics and relevance judgements and then devise a methodology for benchmarking the results of our search engine. In the sections that follow we outline our approach to these problems using a learning-to-rank methodology and detail the comparison to the UCL search engine and google domain-specific search engine.

[1] https://www.funnelback.com/case-studies/university-college-london

Chapter 1

Implementation

1.1 Technology & Approach

One of our objectives was to learn as much as we could about how the components of a search engine work; as such we took the approach of custom-building as much as possible without relying on open source libraries. To achieve this we have compromised on some extended functionality in pursuit of having greater control and understanding of the search engine components. In addition to the basic search engine features we have implemented a learning-to-rank solution even though this was beyond the official scope of the project.

The team members have strengths in different programming languages and different languages themselves have different strengths, therefore various subsystems were programmed in diverse languages where appropriate. The Pagerank and search engine was built in Java. The crawler and indexer were built in Python, as detailed below.

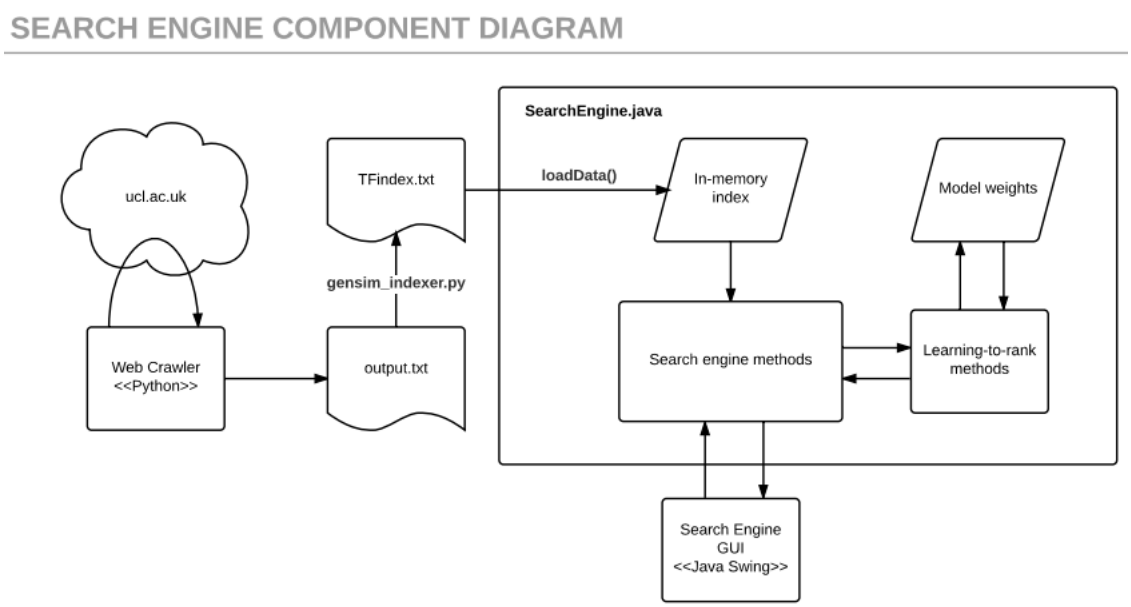

Figure 1.1: The form of a basic search engine

1.1.1 Crawling and Indexing

The initial scope of the project[1] suggested the use of custom elements for the search engine. Rather than using an existing framework, such as **scrapy**[2], the group decided to write a crawler without such aides, as a means to better understand the architecture and behaviour of a web crawler.

The structure of *UCrawL* was loosely inspired by the form of the *Mercator* crawler[3], utilizing a multi-threaded downloader and a url frontier, implemented as a queue. However, due to the exact nature of the project, a number of implementation differences were introduced. The crawler is designed to scrape data from **one** domain only, ignoring all external links. This is in accordance with the coursework description. This did present a small difficulty however, in that to generate sufficient data within a a resonable timeframe, politeness policy[4] regarding repeated requests to the same server had to be largely ignored. Even running in a multi-threaded fashion, DNS latency is still the dominating factor in the efficiency of the crawler: normally this factor could be diffused by walking (psuedo) randomly through the scraped urls thereby issuing requests to a large number of different servers, but this was not the case here due to the monothematic domain. In an attempt to respect some aspects of politeness policy, the crawler searches for and obeys the stipulations listed in the *robots.txt*, located in the markup of many

[1] Project 1 was selected from the coursework document, which called for the creation of a search engine for the UCL website, `http://www.ucl.ac.uk`

[2] `scrapy.org`

[3] Introduction to Information Retrieval, Manning et al, 2008

[4] `https://en.wikipedia.org/wiki/Web_crawler`

(but by no means all) webpages.[5]

The indexer was implemented twice in Python. While the initial version worked[6] reasonably well, it took rather longer to produce a result than was deemed acceptable. To index a collection of c.21000 documents took on average 710 seconds. Further research revealed the existence of the excellent *gensim*[7] library. Rewriting the code resulted in a file of much smaller size (and presumably of greater maintainability) and much greater running speed. Without stemming, indexing was complete in 38 seconds. With stemming[8] resulted in only a slight speed penalty, but much more useful data.

A small number of utility scripts were also written for ease of data manipulation. Due to the incremental, at times tentative, development process, there were occasional incompatabilities between old (but still valid) data, and newer formats. The scripts were designed to ease these issues. There are also a number of scripts to do elementary data analysis, including a human-readable list of term frequencies, and a list of bigram frequencies. These were instrumental in deciding which terms, overly dominant in the collection, could be safely added to a custom stoplist.

1.1.2 The Search Engine

The search engine comprises of three main parts:

- The dataset saved into an index.
- A collection of retrieval methods to query the dataset.
- A learning algorithm.

The raw data is pre-processed to form document term frequency vectors[9]. The engine reads these vectors into a vector space model index to facilitate fast look-up speeds.

Basic term:document associations are saved to a Boolean inverted index, while document:term-frequency vectors are stored to an arraylist ordered by document ID, which can be queried from the inverted index. This allows simple Boolean lookups to be performed on the inverted index, while more complicated retrieval algorithms such as bm25 make use of the document:term-frequency vectors.

[5] As an interesting aside, the `http://www.cs.ucl.ac.uk` does not allow for crawling, an unfortunate situation given that it was listed as a possible target in the coursework paper.
[6] the sourcecode is still present in the github repo and in the assessment zip
[7] `https://radimrehurek.com/gensim/`
[8] using the ntlk library `http://www.nltk.org`, better maintained than the equivalent implementation in gensim
[9] see fig1.1

Table 1.1: **Top 10 pages by PageRank**

www.ucl.ac.uk	0.01769519400935117
www.ucl.ac.uk/accessibility	0.016912985296476814
www.ucl.ac.uk/foi	0.016634130768443414
www.ucl.ac.uk/privacy	0.01659894240968669
www.ucl.ac.uk/contact-list	0.016582308418126135
www.ucl.ac.uk/disclaimer	0.016566870436245235
www.ucl.ac.uk/cookies	0.01637414320466052
www.ucl.ac.uk/staff	0.01415962872136827
www.ucl.ac.uk/prospective-students	0.014085207831860392
www.ucl.ac.uk/students	0.013631117696195717

1.1.3 Retrieval Methods Implemented

The retrieval process first selects all documents that contain any words from the query, then these documents are ranked using the different models outlined below. The scores from each model are then weighted and combined to give an overall ranking.

Boolean AND: This model simply assigns a score of 1 if all query terms appear in the document, and 0 otherwise.

Cumulative term frequency: Simple cumulative count of the total number of times each query term appears in the document.

PageRank: Page rank is calculated using the power iteration method. The code can be found under the pageRank package with 2 classes, PageRank.java and PageRankStart.java. PageRankStart contains the main method which reads in the raw data which consists of a *txt* file containing a visited web page and an associated list of links. 23700 crawled pages with 65000 links in total. PageRank.java arranges this data into an adjacency list consisting of the webpage and its links. A list was chosen as the data structure as a matrix would quickly run out of heap space. Multiple links between pages were excluded and a teleporting factor of 0.15 was included. The power iteration method took 18 iterations and approximately 5 minutes before the probabilities stabilised to within 0.000001.

As expected, *ucl.ac.uk* was top. The rest were also high as they are linked from every other page. The full list of results were output to another .txt file, (PageRankScores) for the search-engine to read.

tf-idf: There are many variations of tf-idf, the version used in our search engine uses the formula given below:

$$TFIDF = \sum \left(\frac{termFrequency}{docSize} \right) * \log_{10} \left(\frac{collectionSize}{documentsContainingTerm} \right)$$

Cosine similarity: Standard Cosine similarity was calculated between the query vector and the document vector using the formula[10]

$$similarity = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

BM25: The code for BM25[11] was only fully completed after the training and evaluation phase of the project and after the learning algorithm had been completed, therefore it was not included in the system. However it could be included for future testing without much extra work. By inspection without evaluation, BM25 seems to provide good results, as would be expected.

[10] https://en.wikipedia.org/wiki/Cosine_similarity
[11] https://en.wikipedia.org/wiki/Okapi_BM25

Chapter 2

Utilization & Training

2.1 Using the search engine

The search engine can be accessed via a simple GUI found in the package SearchEngineGui.[1] There is a text field for user queries and six buttons with the six different search methods described above. Search results are loaded into the textbox upon submission of a query; clicking on links will load the webpage in the user's default browser. In addition, there is a learning to rank search button toward the bottom right. This uses the weighted sum of all search methods as discussed below. The learning to rank search procedure enables checkboxes down on the right hand side of the retrieval results which are selected by the user at each training step to indicate relevant documents. After selecting the relevant documents the user then presses the "learn" button which will recalculate all the model weights (discussed below) and proceed to the next learning step.

2.2 Learning Algorithm

The learning algorithm is based on a weighted sum of all retrieval methods. The method loosely follows the Adaboost paradigm (en.wikipedia.org/wiki/AdaBoost) in which the better predictive retrieval method is progressively weighted more highly after training relative to other models.

Method:

1. The weights of all contributing models are initially set to 1.
2. 30 random novel UCL-related queries are used as a training corpus.
3. For each query the user is asked to select a checkbox on the GUI next to each relevant retrieved result (see fig 2.1).

[1] see fig 2.1

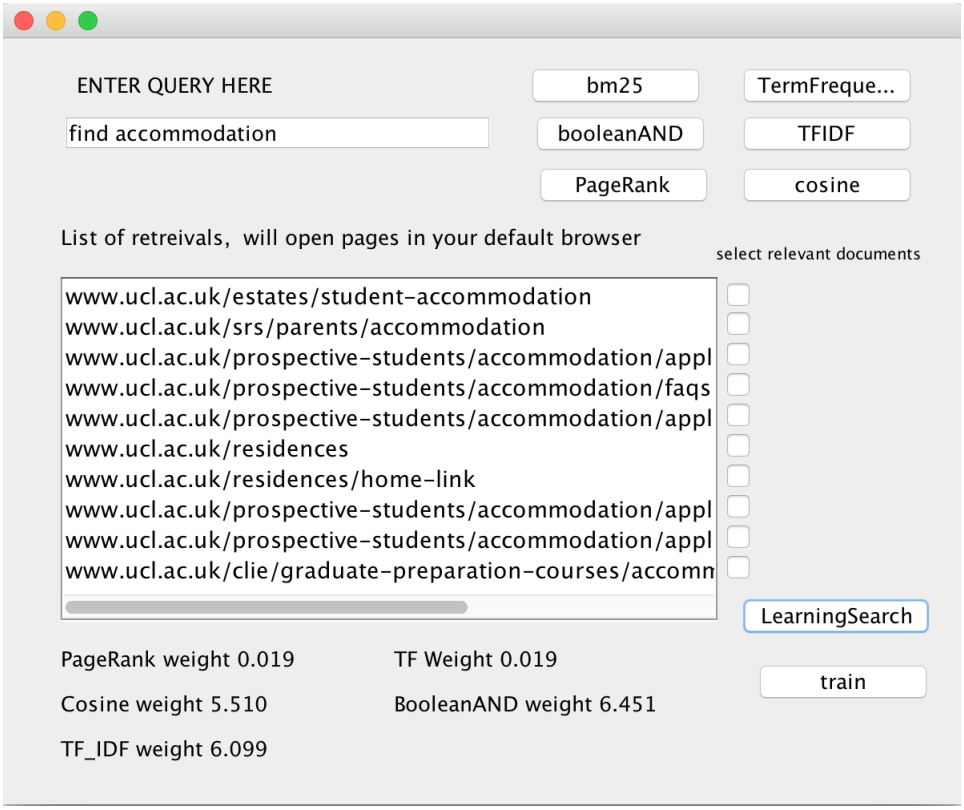

Figure 2.1: The SearchEngineGui

4. A loss function calculates which of the individual retrieval methods performed better or worse, then reassigns the weights as described below. The new weights are logged at each training set for analysis.

Each atomic search method is initially given a weighting of 5. This value was chosen so that changes would be large enough to be significant, but not too large as to cause erratic swings in the model values between learning steps. At each training step the retrieved list of documents and user relevance feedback are used to recalculate the atomic search method weights. This is done for each search method individually by first re-ranking the top ten retrieved documents as they would be ranked using that search method alone. Scores are awarded ranging from 10 to 1 for each relevant document depending on where it would have been ranked (10 for 1st place, 9 for position 2 etc.). New weightings are then assigned by multiplying the old weight by its score divided by the average score of all atomic functions:

$$W_i = W_i * \frac{score_w}{score_{ave}}$$

This means that search methods which assign high rankings to relevant documents have their weighting adjusted upwards, while search methods that assign lower ranks to relevant documents have their

weighting adjusted downwards.

Chapter 3

Results & Evaluation

3.1 Learning Process Results

To implement our learning-to-rank method a 30 step training period was undertaken as discussed above. User relevance feedback was used at each step to adjust the weights of the five contributing atomic search models as can be seen in fig3.1.

The weights of each model changed significantly over the training period. We can conclude from this that the PageRank and Term Frequency models became effectively irrelevant to the aggregate retrieval model in the post-training system. By contrast Cosine Similarity, Boolean-And and Term-Frequency:Inverted-Document-Frequency became roughly equal contributors to the model by the end of the training period.

3.2 NDCG Evaluation

The plot in figure 3.2 allows for a simple comparison of the NDCG@10 values for each of the 20 test queries for the three main IR systems; Google, UCL Search and our post-training system. As can be seen, Google outperforms the other two for the majority of the queries, having the highest NDCG for 15 out of the 20 queries, with the post-training system highest on 3 occasions and UCL search on 2. The plot also shows that the post-training system has higher NDCG@10 scores than the UCL search on 14 occasions with some substantial differences within them, suggesting that, based on our test queries, the post-training system performs better than the UCL search.

We also compared the NDCG@10 for the pre-training and post-training systems. As shown in Figure 3.3, it can clearly be seen that the NDCG scores greatly increased due to the training and adjustment to the parameters that took place. Although, it did not have the same effect for all queries, as it can be seen for query 5 and 18 that the pre-training system outperforms the post-training system.

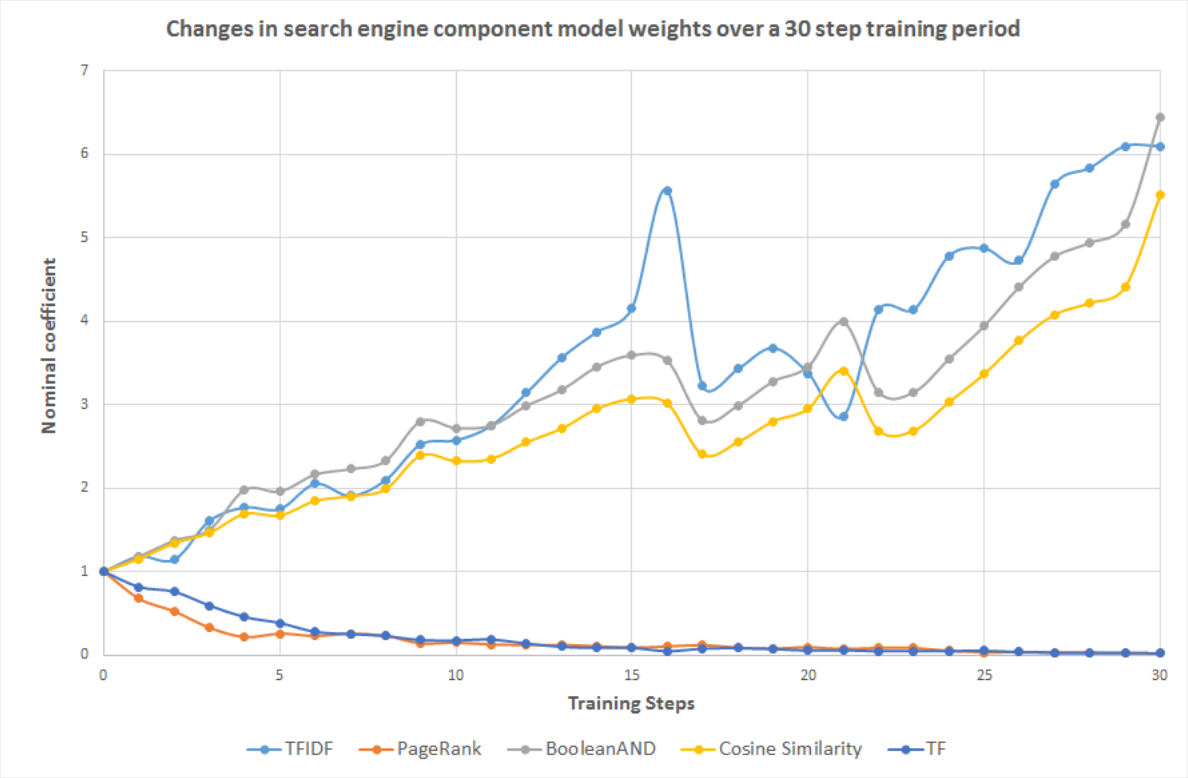

Figure 3.1: Learning results

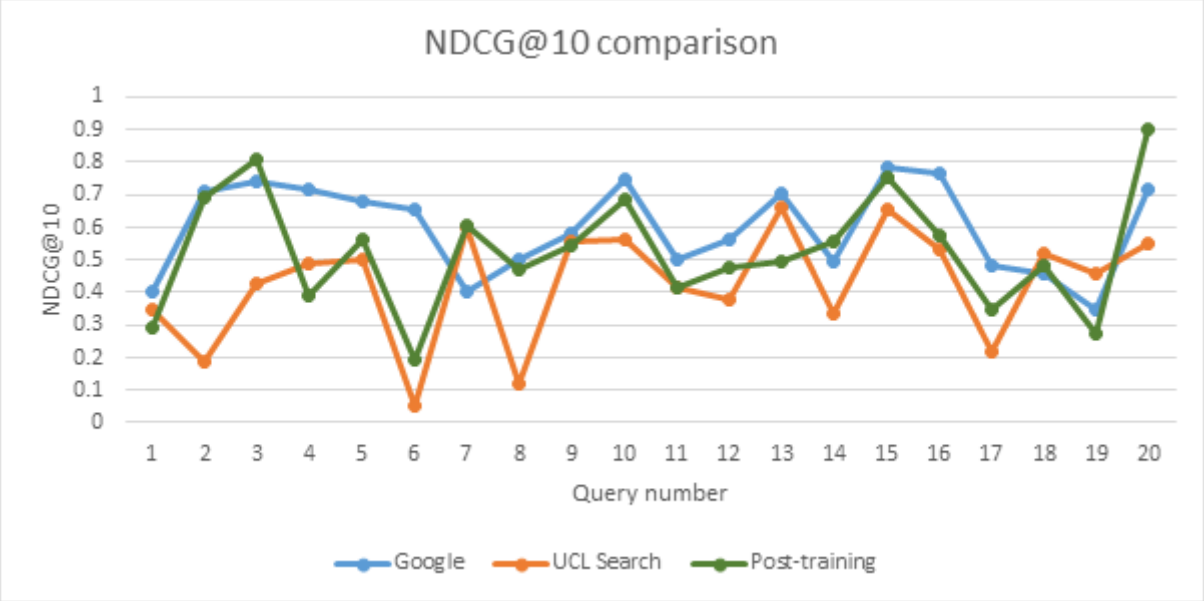

Figure 3.2: normalized discounted cumulative gain at rank 10

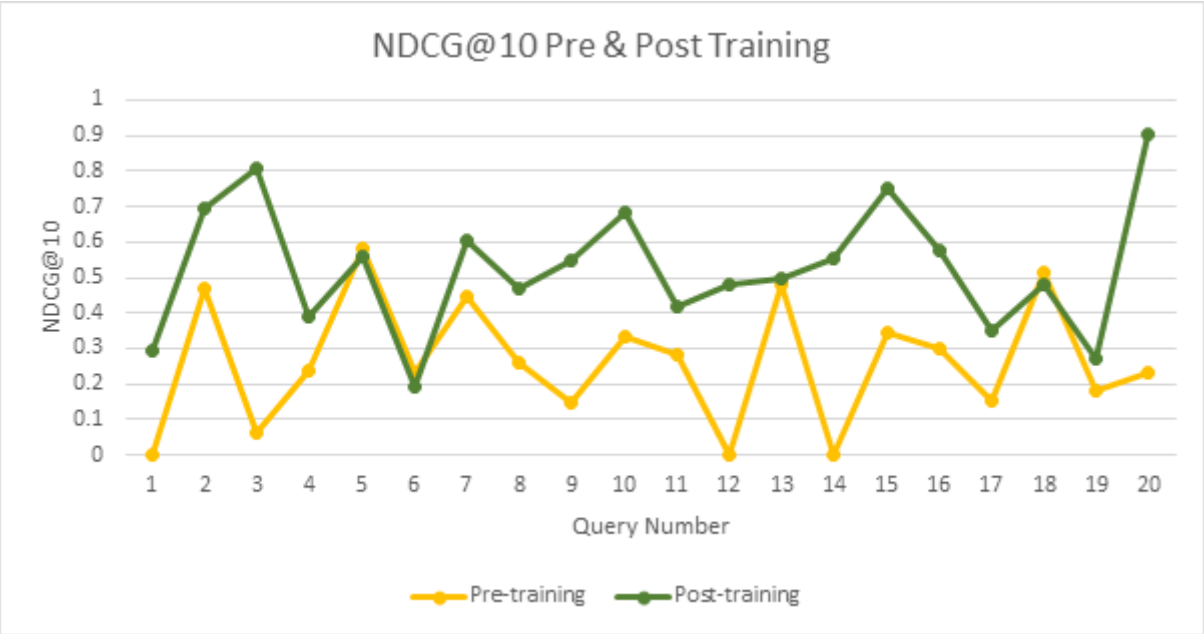

Figure 3.3: ndcg before and after training

3.3 Comparison of mean average NDCG at K

To collect the results, 20 predefined query topics (see query_topics.txt) were submitted to each search engine model. The top ten documents retrieved by each search engine were recorded and manual relevance judgements were made. These relevance judgements and ranking scores were aggregated into a qrels and ranking file respectively for each search engine. These files were then analysed using the NDCGdriver.java and NDCG.java classes to output NDCG at K scores for each query topic, and mean average NDCG at K scores for each search engine. Given that we did not have relevance judgements for a large sample of the document corpus we decided to normalise the DCG scores using maximum ideal DCG scores set to 4 for each query topic. This artifact may have caused the slight negative gradient of the mean NDCG@K lines above, however this should not be significant for comparison purposes. Another consideration that is relevant to comparisons is that the UCL and google domain specific search use the full corpus of ¿ 1 million urls, while our search engine uses a subset of around 21,000 pages. Anecdotal comparison of the search results from google and our post-training model indicates that our results have less diversity. As such the relevance scores could be slightly inflated, given that multiple relevant yet very similar documents are retrieved. To improve this in the future we could introduce a diversity discounting model such as Portfolio or MMR.

Another evaluation measure that we used, was to work out the average precision at each k from 0

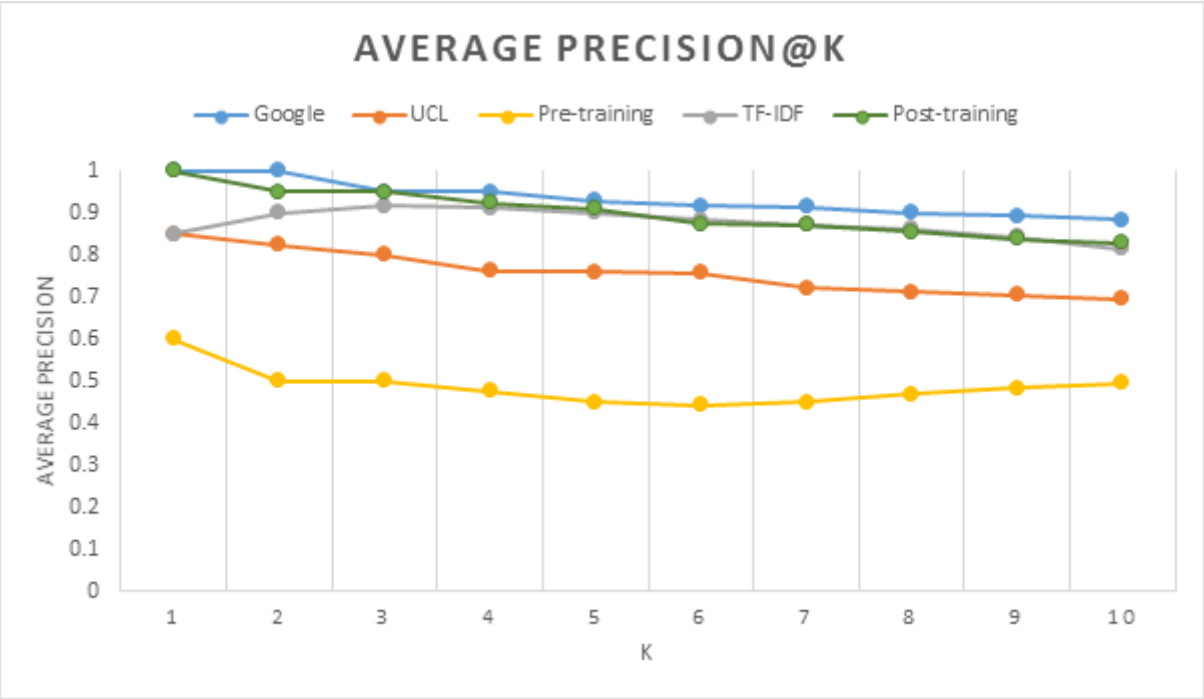

Figure 3.4: average precision

to 10 for each IR system. Precision displays the percentage of relevant documents out of the retrieved documents at each step. Figure 3.4 displays the flow of the average precision as the number of retrieved links increases. It shows that both the Google system and post-training system overall retrieve the highest proportion of relevant documents throughout, with the top result for both always being relevant. It can also be seen that the pre-training system roughly retrieves a relevant document only half of the time, with the UCL search engine being relevant at around 75% of the time. The tf-idf system for the first three documents retrieves less relevant documents than our post-training system but after that, the two retrieve a very similar proportion of documents.

Chapter 4

Benchmarking Performance

To evaluate the results above and benchmark the performance of our post-training IR system relative to existing systems we describe a series of hypothesis tests below. In each test we assume that the effectiveness score differences are meaningful and that the effectiveness score differences follow a normal distribution. We use the NDCG@10 scores for each topic as our effectiveness metric. Each of our tests consists of a null and the alternative hypothesis. We perform various calculations to obtain the mean score differences and calculate the standard deviation by using the corrected sample standard deviation formula:

$$\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

These values then allow the calculation of the t-statistic:

$$\frac{\overline{x}}{\sigma_x}\sqrt{n}$$

The degrees of freedom[1] is found using the formula $\frac{n}{2} - 1$, since we are performing a paired sample t-test. n = 20 here, hence degree of freedom = 9.This all combined enables the use of t-tables to find the corresponding p-value. If this p-value is less than the significance level, chosen here to be 5% throughout.

4.1 Hypothesis Tests

The first test detailed below, tested whether the post-training system is significantly better than the pre-training system at a 5% level:

Null hypothesis H_0: The post-training system is no better than the pre-training system.

Alternative hypothesis H_1: The post-training system is better than the pre-training system.

[1] https://en.wikipedia.org/wiki/Student%27s_t-test

Performing the calculations as described above, we obtain a t-value of 5.209611023 and therefore a p-value of 0.000278 which is ¡0.05, therefore the null hypothesis is rejected and we conclude that the post-training system performs better than the pre-training system.

The second test we performed was whether the post-training system is significantly better than the UCL search:

Null hypothesis H_0: The post-training system is no better than the UCL search.

Alternative hypothesis H_1: The post-training system is better than the UCL search.

We find a t-value of 2.359132971, and obtain the corresponding p-value of 0.021332 which is ¡0.05, hence the result is significant at the 5% level. Therefore we reject the null hypothesis and conclude that the post-training system performs better than the UCL search engine.

The third test we performed was whether the post-training system is significantly better than the Google search:

Null hypothesis H_0: The post-training system is no better than the Google search.

Alternative hypothesis H_1: The post-training system is better than the Google search.

For a test at the 5% level, in order for the result to be significant, the t value needs to be larger than 1.833. But since the t-value obtained is less than this, namely -2.096, we accept the null hypothesis that the post-training system is no better than google.

The final test performed was to establish whether the post-training system is significantly better than the tf-idf system.

Null hypothesis H_0: The post-training system is no better than the tf-idf system.

Alternative hypothesis H_1: The post-training system is better than the tf-idf system.

The t-value obtained is 1.219138 and the corresponding p-value is 0.126897, which is ¡0.05 hence the result of the test is not significant and we accept the null hypothesis that the post-training system is no better than the tf-idf system.

Chapter 5

Conclusion

In addressing the problem of searching and retrieving results from the ucl.ac.uk domain we built a custom web crawler, indexer and evaluation tools. To take our project beyond its original remit we also implemented a learning-to-rank system and trained it with real user relevance feedback. To avoid learning bias we ensured that the training query topics were different to the test query topics. From the benchmarking tests conducted above it can be concluded that our post-training system retrieves more relevant results than the UCL search engine at least across the 20 test query topics. However the results are not significantly better than google or the tf-idf search model in isolation. A caveat consider in these analyses is that the document corpus in our search engine was only a 21000 subset or the ¿ 1 million urls in the *ucl.ac.uk* domain. Furthermore it has been anecdotally noted that while search results are highly relevant in our post-training model, they may lack diversity. Therefore we can conclude that our post-training search engine model performs well in comparison to the existing UCL search and we look forward to implementing it on the site in the coming months.

5.1 Future work

From a software engineering perspective, at present many parts of the project sit, at best, uneasily together. Manual intervention is still required at many stages of the process to convert data from one format to another. This is obviously extremely undesirable, and requires further effort to fully automate the process[1]

Anecdotally, we noticed that the results from our search engine did not have the same level of diversity as the google domain search. To better maximise coverage and minimise redundancy we could introduce diversification models; for example, maximal marginal relevance (MMR)[2] or Portfolio Theory[3]. While learning to rank was outside the official scope of this project, we decided to investigate

[1]Unit testing is also sorely lacking for many parts of the codebase.
[2]http://www.cs.cmu.edu/~jgc/publication/The_Use_MMR_Diversity_Based_LTMIR_1998.pdf
[3]http://dl.acm.org/citation.cfm?id=1571963

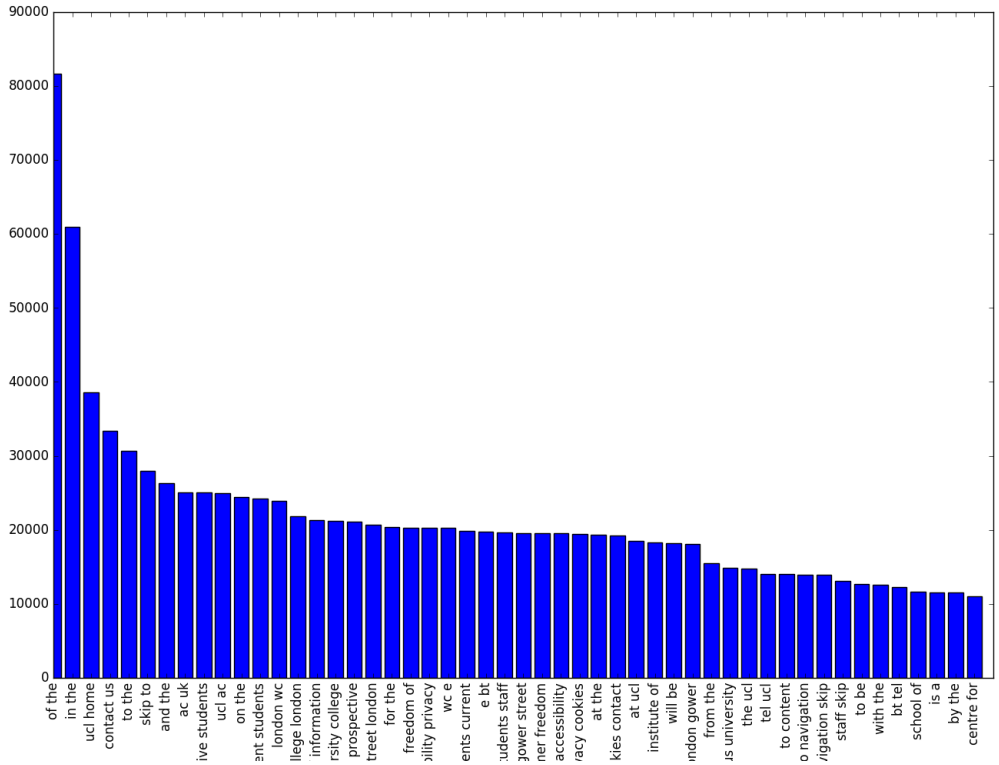

Figure 5.1: The 50 most common bigrams clearly demonstrates the need for a comprehensive and domain-specific stoplist.

it as a logical extension.To take this forward in the future more retrieval methods could be included in the algorithm. BM25 was written too late to be included in this round of evaluation, however we suspect it is very similar to the other retrieval methods and as such it might not give significant improvement. Possible methods to include which could increase the diversity within the results include searching just URL text, searching headers or searching anchor text. This would require changes to the indexing with a specific table relating directly to these areas. Late work analyzing bigrams present in the document collection also suggested a number of different approaches that might be incorporated into a more sophisticated search engine, including the statistical analysis of collocations for possible usage in information retrieval.[4] Certainly a more complete understanding of the dataset would lead to a more tailored stoplist and possible space savings for the postings list[5].

More interesting and useful bigrams occur once the vocabulary has been pruned of "useless" words. It is interesting to note that, at least for the first fifty bigrams, a frequency relationship that might be

[4]Foundations of Statistical Natural Language Processing, C. Manning and H. Schütze, MIT Press, 1999

[5]Understanding the collocations present in the collection for example might result in a *Shannon's Game* type situtation, where strongly collocated groupings could be partially stored due to the high probability of mutual occurence

described as loosely obeying *Zipf's Law*[6] can be observed. This law states that the frequency of a word is inversely proportional to its ranking according to the following formula:

$$P(r) \approx \frac{1}{r \ln(1.78R)}$$

where R is the number of different terms.

[6]"Law" is perhaps an overstatement of the generality of this statement: "observation" might be a more accurate description.

Bibliography

[1] Bird, Steven, Edward Loper and Ewan Klein (2009), *Natural Language Processing with Python.* O'Reilly Media Inc.

[2] Radim Řehůřek and Petr Sojka, *Software Framework for Topic Modelling with Large Corpora* 2010

[3] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, *Introduction to Information Retrieval*, Cambridge University Press. 2008.

[4] Christopher Manning and Hinrich Schütze, *Foundations of Statistical Natural Language Processing*, MIT Press. Cambridge, MA: May 1999

[5] Jaime Carbonell, Jade Goldstein *The Use of MMR, Diversity-Based Reranking for Reordering and Producing Summaries* `http://www.cs.cmu.edu/~jgc/publication/The_Use_MMR_Diversity_Based_LTMIR_1998.pdf`

[6] Jun Wang, Jianhan Zhu *Portfolio Theory of Information Retrieval* `http://dl.acm.org/citation.cfm?id=1571963`