# CS400 — Problem Seminar — Fall 2000
# Assignment 4: Search Engines

Handed out: Wed., Oct. 18, 2000
Due: Wed., Nov. 8, 2000
TA: Amanda Stent (`stent`)

**Note: You have 3 weeks for this assignment, rather than 2, so that you will also have time to work with an advisor on your term project proposal (due Oct. 27). But try to start this assignment before then, especially since I'll be out of town Oct. 30–Nov. 7.**

## 1 Introduction

When you type a query into a search engine, you get back a ranked list of "relevant" documents. But how does the search engine measure relevance? And how does it find the relevant documents quickly?

This search engine task is sometimes called "ad hoc document retrieval." It is the classic problem (though not the only interesting one) in the burgeoning field of **information retrieval** (IR). In this assignment, you'll get to try your hand at making a search engine better—as the search engine companies are continually trying to do.

As always, the assignment is somewhat open-ended: show us what you can do with a real engineering problem. Can you come up with a clever, original approach? Can you make it elegant, and can you implement it and evaluate how well it works? We will see whose approach has the best performance!

This assignment will also force you to find resources that will help you. You will probably want to browse through some IR papers to get a sense of what is likely to work. And in order to do well, you will probably have to perform some non-trivial operations on the text. Unless you want to reinvent the wheel, this means tracking down someone else's software tool and figuring out how to download, install, and use it. Of course, you are welcome to ask for advice!

There are many clever ideas for making search engines faster and more accurate. Some of them are in the published literature. Others are secrets of the search engine companies, but you can find out a fair amount about how search engines work by experimenting with them. Others haven't been thought of yet, but you may be able to think of them!

Note: An interesting research move is always to redefine the problem. Why should the search engine behave like a function that turns a short query into a ranked list? Perhaps there should be more negotiation with the user, or more informative output than a list of entire documents. If you are interested in altering the problem that you will solve, come talk to me about it first.

## 2 Annotated Data

As in our vision project, we will be using a training-and-test paradigm. You will have a collection of documents to index (no need to crawl the web; they'll be stored locally). You will also have a set of training queries to help you build your system, and a set of test queries to help you evaluate it. For every query (in both training and test data), an annotator has provided a list of the "true" relevant documents.

A collection of documents is called a *corpus*. The plural is *corpora*.

The corpus we will work with comes from the first four TREC competitions. (TREC is the "Text REtrieval Conference.") The total dataset for those competitions contains over 1,000,000 documents. We will be working with a subset of about 200,000 documents, 150 training queries, and 50 test queries. For each query, the annotators have manually chosen an average of 200–250 of the documents as relevant. (They didn't consider all 1 million documents: they used automatic methods to narrow down the document set to about 2000 possibly relevant documents per query, and judged those by hand.)

The data are in `/s28/cs400-ir/train/docs`. You can find out more about how to look at this corpus—under the name `trec`—in `/s28/cs400-ir/README`. You may find it useful to know that `gzcat` is a Unix command that works just like `cat` except that it expects compressed input and produces uncompressed output. It is equivalent to gunzip -c.

For understanding how different systems work, you may want to experiment with extremely small, artificial corpora, such as `tiny`, in `/s28/cs400-ir/train/tinydocs`. You will probably also want to save time by working with medium-sized corpora, such as `trec_ap`, some of the time.

# 3 Getting Started

In class, we will cover a simple information retrieval paradigm known as TF IDF. The TF IDF approach treats each document like a bag of words in no particular order. Our starting point will be the Managing Gigabytes (MG) system. See `/s28/cs400-ir/README` to get started—I have done some work so that you will be able to run the basic system on this corpus easily and quickly.

An alternative but less convenient starting point is Andrew McCallum's `arrow` program, which implements TF IDF and several variants that also have the bag of words property. You can find it as `/s28/cs400-ir/bow/bin/arrow`. Try it out:

```
arrow -i dir
```

will create an index (stored in `~/.arrow` by default) of all the documents in directory `dir`, and then

```
arrow -q
```

will let you query this database as if it were a search engine, giving you back the filenames of the top 10 documents. For many more options, type

```
arrow --help
```

If you look at the `main()` function[1] in the source code file `/u/jason/teach/cs400/hw4/bow/src/arrow.c`, you will see that `arrow` is not doing very much work at all. It relies heavily on McCallum's efficient bag-of-words C library, `bow` (also called `libbow`), which is now distributed with some versions of Linux. (`bow`, together with `arrow` and some other simple programs that use it, is installed under `/u/jason/teach/cs400/hw4/bow`.)

Your first step should be to experiment with the different options of MG or `arrow`, using annotated training data (see above). What combination of options works best? Mathematically, what do those particular options make the program do?

Answer the above questions in your writeup. To figure out what the options do, you may have to read the documentation (MG has better documentation than `arrow` or `bow`), or read the code, or use the web and textbooks to figure out what all the terms mean. This is good practice for real life!

[1] Most of the rest of the source file simply interprets the command-line options

# 4 Error Analysis

Your next step should be to study the behavior of MG or `arrow`. What kinds of mistakes does it make on the training data? What would it need to know, or do differently, in order to avoid those mistakes? Discuss in your report.

# 5 Innovating

Bearing your error analysis in mind, now your job is to improve the performance of MG or `arrow` on training data! Try to find *one* plausible and moderately interesting technique that helps at least a little bit, and see how far you can push it.

Here are some things that you might try:

- Correct or normalize spelling. Be case sensitive only when appropriate (you may need special handling for the start of sentences, headlines, etc.).
- Do morphological "stemming": this turns `computing` into `compute`, and perhaps turns `went` into `go`. (Actually, this turns out to be a built-in option to both MG and `arrow`, so you should have evaluated it already; but maybe you can use a more sophisticated stemmer.)
- Try to disambiguate words. Is `lead` being used as a noun or a verb? Does the query mean `Jordan` the basketball player or `Jordan` the country? Disambiguation is needed when one word has two meanings.
- Use some kind of thesaurus, such as WordNet, so that if the query says `Middle East`, you will be able to find documents that mention `Jordan`. Thesauri are needed when two words have one meaning. You could also try to generate a rough thesaurus automatically by statistical techniques; there is a lot of work on this, notably Latent Semantic Indexing.
- Change the distance function (metric) that determines how close a query is to a document. (This can get fancy: for example, you might cluster the documents first and relocate the origin to an appropriate cluster centroid.)
- Find a better way to weight or smooth the word counts.
- Instead of treating the document as a simple bag of words, take the order and proximity of words into account. For example, index phrases as well as words.

- Improve the handling of names. A person's name may appear in different forms (last name first, initials, etc.). The same is true for company names: `IBM` should match `International Business Machines Inc.`
- In our training and test data, queries will be actual English questions. So pay attention to the form of the query. If it starts with `What city`, you would like to find a document that mentions some city in the appropriate context. If it starts with `When`, you want a date. There are "named entity" tools that can discover cities and dates for you.
- Use some kind of parser or partial parser to extract noun phrases that should be recovered as some kind of a unit: `gas pump`, `information technology`. Or do this for other cases where words are crucially modified by other words: `launch` ... `rockets`.
- Weight certain parts of a document more highly than others: e.g., the title, the lead paragraph, etc.
- Do documents have properties that make them more likely to be relevant to queries *in general*—or at least, to the *kind* of queries used in this test? Should such documents be ranked higher? (As discussed in class, Google uses hyperlinks for this.)
- Would it help to exploit the fact that the documents are grouped into directories by source and year? How about other document classes that can be found automatically?

A few of these suggestions would require modifying MG or `arrow` a little bit, or calling some of the other functions in the `bow` library.

But a surprising number can be accomplished simply by applying some transformation (such as stemming or disambiguation) to both the document and the query, and then using MG or `arrow` on the transformed texts. In other cases, you may be able to change the query automatically to a better query (perhaps by adding words or doing some linguistic processing); this is called *query expansion*, and does not involve changing the documents.

Some transformation or expansion techniques call for doing some linguistic preprocessing. Fortunately, you can use someone else's stemmer (for example) just as you can use someone else's edge detector. Most of these problems are well-studied. There are papers and textbooks to read, and there are software tools to handle many of the hard parts of the approaches above. (The tools aren't perfect, but they don't have to be for this task!) A good place to start looking for tools is the Natural Language Software Registry

(http://registry.dfki.de/). Be warned that some tools are tricky to install and learn.

For efficiency, a *two-pass* strategy can sometimes help. Use some kind of a crude first pass that retrieves (say) 3000 potentially relevant documents (high recall, low precision). Then you can apply a more sophisticated strategy to this much smaller dataset.

Perl is your friend when it comes to matching patterns and making changes in text.

By the way, if your solution is to transform the data, you might also try feeding your preprocessed data into someone else's system! For example, you correct the spelling, the next person does stemming on your output, and the result goes into MG or `arrow`. This combination could have a better result than either technique separately, so it could help you win in the testing phase.

# 6 Readings

You will certainly find it helpful to read the Overview of the Eight Text REtrieval Conference (TREC-8), http://trec.nist.gov/pubs/trec8/t8_proceedings.html. Pay special attention to sections 1–2, 3.1, 3.3, 4.1, 5.1. You will be doing the same task as in 5.1: short ad hoc queries, no manual intervention, evaluated by mean average precision.

That overview mentions other TREC proceedings papers that describe individual systems in detail. They are available at http://trec.nist.gov/pubs.html. The proceedings of ACM SIGIR may also be useful.

I have left two books in the lab: *Managing Gigabytes*, a textbook that explains and describes the MG system in great detail, and an excellent paperback collection of papers called *Readings In Information Retrieval*.

You can find out more about TF IDF in the classic books of Gerard Salton, who essentially created the field more than 30 years ago. These books are available in Carlson Library. You might also check out *Modern Information Retrieval*, by R. Baeza-Yates and B. Ribeiro-Neto.

You may also find interesting stuff by searching the web or the library, as always.

You are also free to ask questions of the TA, or me, or anyone else in the department who works on language. We may be able to point you to useful tools or papers. And it's wise to bounce your ideas off people with experience before you put them into practice.

# 7 Testing

Report the performance of your system on test data. If you combined your system with someone else's, as described above, then report the results of your technique and the other person's technique when used separately, as well as the results of the combined system.

This project is based on the TREC ("Text REtrieval Conference") competitions. Just as in TREC, we'll compare all of your systems and declare a winner. So when you're designing, think about what will make your system do as well as possible!

# 8 Reports

Reports are as usual. Write separately, have a friend check whether your English is clear and correct, and remember that you are writing for an audience that *does not know what the problem is*.

Again, don't leave reports till the last minute. Each major section of this handout (Intro, Getting Started, Error Analysis, Innovating, Testing) can correspond to a section of your report. So you can write up each section as you are completing the work on it, if not sooner.

Please don't skip any sections. There are specific instructions earlier in this handout about things that you must include in the report.

Make sure to cite the *Managing Gigabytes* textbook if you use MG: `http://www.cs.mu.oz.au/mg/`. Or make sure to cite McCallum's programs if you use them: `http://www.cs.cmu.edu/~mccallum/bow/` explains how.

Have fun!