Building a Simple Search Engine

Aaron Blumenfeld

January 19, 2019

1 Jaccard Similarity

Given two sets A, and B, the Jaccard similarity of A and B is defined to be

$$\text{Jacc}(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

In the context of search engines, A and B will be sets of words that a document or search query contains. For example, given the search query "computer science courses", and the document "computer organization computer textbook", let's compute the similarity.

The set A would be {computer, science, courses}, and the set B would be {computer, organization, textbook}. Note that `computer` appears twice, but with sets, we don't care about duplicates, so the set B only contains three words.

So

$$\text{Jacc}(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|\{\text{computer}\}|}{|\{\text{computer, science, courses, organization, textbook}\}|} = \frac{1}{5}.$$

Observe that the Jaccard similarity will always be a number between 0 and 1 (since $A \cap B \subseteq A \cup B$ for any sets A and B). A similarity of 0 would mean the two sets are disjoint: there are no words in common. A similarity of 1 would mean the sets are exactly the same, so the original strings are very similar (possibly different word order, or duplicate words). Higher values generally mean "more similar."

As a different example, suppose our strings are "dogs chase cats" and "cats chase dogs." In this case, the Jaccard similarity is 1, so the strings are very similar. Indeed they are, but they don't capture the significant difference in meaning. In practice, Jaccard similarity is a *very* simple idea that can capture similarity quite effectively, but possibly not good enough for more complicated situations (i.e., we are using what's called the *bag-of-words model*).

2 Basic Search Engine (`searchEngine1.js`)

Our application is a toy one: find an episode title (plus season and episode number) of an episode of *It's Always Sunny in Philadelphia* using "fuzzy search" (i.e., inexact string matching) by way of Jaccard similarity. The algorithm is simple: compute the Jaccard similarity between the input and every episode title (which would be stored in a database, .csv file, etc.). Return the episode with the highest Jaccard similarity. When doing this, we also ignore the words "the", "a", and "an." (These are called *stop words*, and real-world stop word lists would likely include additional words.)

This works pretty well. For example, "**water park**" correctly gives "**The Gang Goes To A Water Park**." Unfortunately, "**waters parks**" produces an error (similarity of 0) since water is not the same as waters and park is not the same as parks.

3 Adding NLP (`searchEngine2.js`)

We can fix this by adding NLP. There are a couple different NLP techniques to use:

(1) Stemming. Reduce verbs to infinitives, nouns to singular (or actually stems which are likely prefixes of singular/infinitives). For example, *am*, *are*, and *is* would map to *be*. *Box* and *boxes* would map to *box*.

(2) Lemmatization. This is a lot more complex (and slower), and takes into account part-of-speech tagging and context. For example, *saw* might map to either *see* or *saw*, depending on the context.

In my experience, lemmatization has given slightly better results when combined with Jaccard, but is also slower.

NLP does fix the problem, but the calculations are really slow, and it takes a good second to do the search for each query (at a **small** scale). How can we hide the latency?

4 Pre-compute NLP for Document List (`searchEngine3.js`)

We can pre-compute the NLP calculations on page load so we only need to do the NLP calculation for the one query string whenever we do a search. This mostly hides the latency, but it's still there whenever we load the page.

We can actually change the script tag in `index.html` from `<script src=‘‘searchEngine3.js’’></script>` to `<script src=‘‘searchEngine3.js’’ async></script>`, and it will load the UI immediately. There will still be a problem if the user clicks Search before the pre-computation is complete (doesn't work in Firefox...).

5 Store NLP calculations in a database

We can also pre-compute the NLP calculations and store them in a database and retrieve by making a call to the database. In this simple application, an easy way to simulate this is using `JSON.stringify` and `JSON.parse`. (Type `JSON.stringify(nlp_episodes)` in the console, paste that in as a variable in the code, and type `nlp_episodes = JSON.parse(variable_name)` in the code.)

6 Real-Life Applications

(1) Plagiarism detection. We might want to split the papers into paragraphs and compare each paragraph from paper A to each paragraph from paper B to see if any two paragraphs are suspiciously similar.

(2) Suppose you're a software engineer on a team with a lot of tickets. You might want to find the top 5 or 10 tickets that an incoming ticket is most similar to from the past few months in order to quickly find the root cause. We could do the same thing here and build a simple search engine in a couple hours of work (plus however long it takes to onboard to ticket-searching APIs).

7 More Complex Search Engines

This won't take typos into account. One might consider introducing edit distance for that. More sophisticated systems would also take into account a more extensive stop word list, word frequencies, and so on. If you take an *Information Retrieval* course, you'll learn about systems called *tf-idf* and *BM-25* which are more effective (and more complicated). These systems are very effective at taking into account word **frequency**, which Jaccard does not.

Nevertheless, for small applications, Jaccard similarity works remarkably well, and can be implemented in an hour or two (though one issue with Jaccard is that it skews toward shorter documents — the fewer words in the document that are not in the query string, the greater the chance of matching).

8 Source Code

https://github.com/ablumenf/search-talk