Information Retrivel: Lab 1

Shifei Chen

1 Inverted Index

The number of dictionary dropped from 4286 to 3684 after lowering cases. This is expected as there were many words in `index.txt` in both their captialized and non-captialized form. The number of postings didn't change and remained at 1000, since we didn't remove any of the words the posting list should be the same.

The postings didn't change after removing the top 10 most frequent words from the index either. It indicated that we have an evenly distributed posting list and the postings didn't cluster around several specific terms. Otherwise the number of postings should be smaller.

2 Boolean Queries

I choosed skip pointers as my optimazation to boolean queries and in most cases it reduced the number of comparsions needed in the search. For example for query "really AND kids AND school" the program compared 111 times before optimazation. With skip pointers it only needed 56 comparsions. I have also run a few more times with other queries, the result is showed in Table 1.

Table 1: Number of Comparisons for Different Queries

Queries	**Without Skip Pointers**	**With Skip Pointers**
really AND kids AND school	111	56
really AND class	122	22
school AND class	15	9
you AND me	916	916
me AND .	987	987
class AND you	873	89
me AND class	381	40
me AND left	401	203

Skip pointers didn't always guranteen a smaller number of comparsions. As the result from "you AND me" and "me AND ." showed. In both cases, the terms had very frequent words ("you" and "." were two of the top 10 most frequent words. "me" appeared 385 times in the index file, not one of the most frequent words but still quite often). Query "class AND you" cost just a little bit more than 1/10 of the original algorithm after optimazation and "class" was a rare word in the documents (it only appeared 2 times). In other cases, skip pointers usually saved half of the time in comparsions. The result suggested that if most of the terms were highly frequent in the documents, skip pointers may not appear to be more efficient than the non-optimized version.

Something extra to mention here is that I didn't follow exactly the intersect algorithm with skip pointers as simulating a linked list in Python prefectly is too complicated. I did a simplified version which utilized list index as its pointers.

3 Ranked Queries

By definition in order to calculate $tf\text{-}idf_{t,d}$ of word t and in document d, we need to calculate both the term frequency $tf_{t,d}$ of that word t in document d and the inverted docuemnt frequency $idft$ of t.

So start with the term frequency for word "school" in all three docuements returned by binary search "school AND kids AND really", it's easy to obtain the result from `sort -t ' ' -k1,1 -k2,2n | uniq -c`. The first column is the frequency we were looking for.

$$
\begin{aligned}
tf_{school,72} &= 2 \\
tf_{school,224} &= 1 \\
tf_{school,385} &= 2
\end{aligned}
$$

Then df_{school} could be found after executing binary search for term "school", which was 15, hence we could get the idf_{school}.

$$
\begin{aligned}
df_{school} &= 15 \\
idf_{school} &= \log \frac{N}{df_{school}} \\
&= \log \frac{1000}{15} \\
&= 1.824
\end{aligned}
$$

Finally following the definition we could get the $tf\text{-}idft, d$ for term "school" in all three documents.

$$
\begin{aligned}
tf\text{-}idf_{school,72} &= tf_{school,72} * idf_{school} \\
&= 2 * 1.824 \\
&= 3.648 \\
tf\text{-}idf_{school,224} &= 1.824 \\
tf\text{-}idf_{school,385} &= 3.648
\end{aligned}
$$

Similar to above, we have the other TF/IDF values.

$$
\begin{aligned}
tf\text{-}idf_{kids,72} = tf\text{-}idf_{kids,224} = tf\text{-}idf_{kids,385} &= 1.699 \\
tf\text{-}idf_{really,72} = tf\text{-}idf_{really,224} &= 0.917 \\
tf\text{-}idf_{really,385} &= 1.834
\end{aligned}
$$

So, if we use these three keywords as the bag-of-words represention of these three documents, we could build vectors with their TF/IDF values like so.

$$
\begin{aligned}
\vec{v}(d_{72}) &= [3.648, 1.699, 0.917] \\
\vec{v}(d_{224}) &= [1.834, 1.699, 0.917] \\
\vec{v}(d_{385}) &= [3.648, 1.699, 1.834]
\end{aligned}
$$

And the cosine similarity of each of the two documents are

$$
\begin{aligned}
sim(d_{72}, d_{224}) &= 0.948 \\
sim(d_{72}, d_{385}) &= 0.979 \\
sim(d_{224}, d_{385}) &= 0.956
\end{aligned}
$$

They suggested these documents should be very similar to each other. So after looking at the original document, they did share a few sentences in common. In additon, the sentence "Well , in sh ... school ..." appeared in both document 72 and 385 but not in document 224, which explains why the similarity score between d_{72} and d_{385} was the highest.

4 Crawler

I crawled car ads from blocket.se, spcifically the cars from Stockholm for simplicity (looks like it's not banned to do so in their `robots.txt`). The basic process is like find the element I'm intersted in, e.g. the item title, inspect it in Chrome to see what kind of tag it is and what kinds of characters (attributes, css classes, parent tags, etc.) it has.

One thing worth to mention is that some of the attributes inside the tags were not standard html attributes, for example `itemprop`. And sometimes these attributes can hold multiple values. It looks like BeautifulSoup didn't support extracting multiple values from customized attributes, unlike standard attributes such as `class`. Therefore I ended up with regular expressions.