Eine selbstgemachte Search-Engine

M150 - E-Business-Applikationen anpassen
Vertiefungsarbeit
Fabian Bächli
28.10.18

Im folgenden Dokument werde ich die Suchmaschine, welche ich für das Modul M150 entwickelt habe, genauer erklären.

Was

Bei meinem Endprodukt handelt es sich um ein Java Programm, welches 500'000 Wikipedia-Artikel indexiert. In der Theorie könnten noch mehr Artikel indexiert werden, jedoch reichten in diesen Fällen die 12GB zugewiesener RAM der JVM nicht aus. Der Index wird in ein File gespeichert und kann danach wieder geladen und abgefragt werden. Die von mir gewählte Datenstruktur erfüllt diese beiden Kriteren:

- Sie ist deutlich kleiner als der zu-indexierende Dokumenten-Cluster
- Sie ist schnell durchsuchbar

Der Index ist ein sogenannter reverse-index. Die technischen Hintergründe dazu später.

Wie

Daten

Für meine Zwecke benötigte ich eine grosse Menge semi-strukturierter Daten. Ich wollte die Daten ausserdem in eine Datenbank importieren, um mir den weiteren Umgang mit diesen so einfach wie möglich zu machen. Ich entschied mich dafür, an der deutschen Wikipedia – da diese wesentlich kleiner als die auf englisch ist – mein Glück zu versuchen.
Wikipedia stellt die Daten zur gesamten Wikipedia hier: https://dumps.wikimedia.org/dewiki/latest/ zur Verfügung. Ausserdem gibt es ein Tool namens mwdumper, welches die komprimierte Wikipedia in eine SQL Datenbank entpackt/importiert.
Als ich die Daten nach einigen Fehlimporten, auf 0.4 MB freier Speicherplatz gefüllte Festplatten und einigen Missgeschicken meinerseits endlich in MySQL hatte, begann ich damit, Queries zu schreiben, welche mir die Daten in der von mir gewünschten Form lieferten. Dies stellte sich als garnichtmal so einfach heraus, denn, eine Dokumentation zum Datenbankschema, sprich, welche Tabelle für was ist, welche Spalten welche Werte beinhalten usw. scheint es nicht zu geben. Nach einigem Proben und Herumversuchen hatte ich einen Ansatz, welcher es mir ermöglichte, alle Artikel-Titel und deren Inhalt abzufragen. Nun stellte sich aber heraus, dass Wikipedia zu einem Grossteil aus Begriffserklärungsseiten zu bestehen scheint, jedenfalls schienen mir diese weitaus mehr verbreitet als die klassischen Wikipedia-Pages, die wir alle kennen und

lieben. Also musste ich mein Query so bauen, dass es diese, und andere, für mich ungewollte, Seiten filtert.
In Java stellte ich die Verbindung zu Datenbank dann mit dem JDBC Treiber her, was reibungslos funktionierte.

Indexieren

Zuerst wollte ich die Indexierung mit einem Framework bewerkstelligen, entschied mich dann aber dafür, lieber etwas Eigenes zu erarbeiten.
Als erstes setzte ich mich mit der dem Indexierungsprozess zugrundeliegenden Theorie auseinander. Grundsätzlich gibt es hierbei zwei Methoden, welche mir interessant schienen: das Mappen von Dokumenten im high-dimensional-space und der reverse-index.
Der erste Ansatz erstellt aus Dokumenten Vektoren, welche ein Produkt aus den Wort-Vektoren sind. Zum Erstellen der Wort-Vektoren (der Prozess wird Embedding genannt) kann man ein vortrainiertes Model, zum Beispiel von hier: https://nlp.stanford.edu/projects/glove/ verwenden. Dieser, in den Bereich des Natural Language Processing fallende Ansatz, wäre für mich persönlich am spannendsten gewesen, da er ein hohes Theorieverständnis erfordert. Leider hatte ich schon zu viel Zeit mit dem Importieren in MySQL verdödelt, also nahm ich den schnelleren, klareren Weg: der reverse-index.
Der reverse-index funktioniert so:

- Man erstellt einen **forward-index** in dieser Form:

```
{
 "document_1": [
  {"term": "house", "count": 4, "tf_idf": 1.2},
  {"term": "car", "count": 9, "tf_idf": 0}
 ], "document_2": [
  {"term": "cat", "count": 5, "tf_idf": 1.5},
  {"term": "car", "count": 6, "tf_idf": 0}
 ]
}
```

    - Der forward-index bezeichnet das Mapping der einzelnen Terme in einem Dokument auf das Dokument. Es wird jeweils gezählt, wie oft der Term im Dokument vorkommt.
    - Das tf_idf Property steht für «term-frequency inverse-document-frquency» und wird berechnet, indem man die absolute Häufigkeit eines Terms von allen Dokumenten (idf) mit der relativen Häufigkeit des Terms in einem Dokument (count) multipliziert. Der Wert kann erst berechnet werden, wenn der reverse-index erstellt wurde.

- Nachdem man den forward-index erstellt hat, kann man den **reverse-index** erstellen, welcher so aussieht:


```
{
 "house": {
  "idf": 0.3,
  "documents": [
   "document_1"
  ]
 }, "car": {
  "idf": 0,
  "documents": [
   "document_1",
   "document_2"
  ]
 }, "cat": {
  "idf": 0.3,
  "documents": [
   "document_2"
  ]
 }
}
```


    - Der reverse-index ist das Mapping von Dokumenten auf Terme.
    - Das idf Property steht für «inverse-document-frequency», die Umgekehrte Dokumenthäufigkeit also. Die Zahl ist tief, wenn ein Wort häufig ist und hoch, wenn es selten ist. Wenn ein Dokument einen hoch-spezialisierten Term enthält, ist die Chance gross, dass dieses Dokument das gesuchte ist, wenn dieser Term im Query eingegeben wird. Dies ist die Formel dafür:
      $\log_{10} \frac{N_D}{f_t}$.
        - $N_D = Anzahl\ Dokumente$
        - $f_t = Anzahl\ Dokumente, welche\ den\ Term\ t\ enthalten$

Ich bastelte also eine Java-Implementation davon, welche im Wesentlichen genau aussieht, wie die oben abgebildeten JSONs, aber halt mit HashMaps und ArrayLists. Den forward- und reverse-Index speicherte ich ausserdem als File, da ich diese nicht jedes Mal, wenn ich das Programm starte erstellen will.
Um den forward- und reverse-Index erstellen zu können, musste ich die Result des MySQL Queries tokenizen. Dazu iterierte ich über jedes Wort aus den Resultaten und erstellte zuerst den forward-, dann den reverse-Index. Die Header der Artikel splittete ich in einzelne Wörter und zählte jedes Wort als 50-mal vorgekommen (Term-Weighting), da ein Wort, welches im Titel eines Artikels vorkommt eine sehr grosse Chance hat, der Suche zu entsprechen. Jedes Wort normalisierte ich (lowercase, entfernen von non-ascii chars usw.), bevor ich es speicherte. Man könnte hierbei noch weiter gehen und zum Beispiel den Porter-

Stemmer-Algorithmus anwenden, um die Wörter in ihre Grundform zu bringen. Hätte ich mehr Zeit gehabt, hätte ich an der Normalisierung der Daten, sowie dem Term-Weighting mehr Zeit investiert, da diese beiden Dinge ziemlich rudimentär implementiert sind.

Querien

Das Query-String wird zuerst normalisiert, dann bei den whitespaces gesplittet. Die einzelnen Query-Wörter (keys) werden dann im reverse-index gesucht. Die Resultate werden nach der Grösse ihres tf_idf-Werts geordnet.
Eine grosse Schwachstelle, welche dieser Ansatz hat, ist, dass ein forward-index nur genau dann gefunden wird, wenn der Key genau matcht. Es kann nicht mit Wahrscheinlichkeit gearbeitet werden, wie dies beim Dokument-Vektor Ansatz möglich gewesen wäre. Man könnte dieses Problem wiederum mit der radikaleren Normalisierung der Daten weniger gravierend machen.

Schwachstellen

- Daten werden zu wenig normalisiert
- Die tf_idf Werte bei Queries mit Whitespaces werden noch nicht untereinander verglichen. Somit wäre es möglich auf ein Query mit Whitespaces, eine Liste von Resultaten anzuzeigen, und diese nach ihren Wahrscheinlichkeiten zu gliedern, anstatt mehrere Listen zu den einzelnen Wörtern.

Reflexion

Gutes

- Theorieauseinandersetzung
- Implementierung
- Sehen von Schwachstellen
- Viel Neues gelernt (selber implementiert, obwohl es Frameworks gäbe)

Schlechtes

- Zuviel Zeit am Datenimport verbraucht und somit zu wenig Zeit gehabt, eine spannendere Implementation zu versuchen