Praktikum Information Retrieval
Einführung

Matthias Jordan

November 2011 — Januar 2012

1 Einführung

1.1 Ziel der Veranstaltung

Die Teilnehmer des IR-Praktikums sollen lernen, die in der Vorlesung vorgestellten Konzepte an einem praktischen Beispiel anzuwenden. Im Rahmen des Praktikums besteht darüber hinaus die Gelegenheit, Methoden der Softwareentwicklung auszuprobieren oder einzuüben.

1.2 Ablauf

Das Praktikum ist in drei Blöcke aufgeteilt.

Im ersten Block steht die rudimentäre Implementierung eines Suchsystems auf Basis von Lucene im Vordergrund. Entwicklungsziel ist dabei, eine gegebene Kollektion zu indexieren und Suchen darauf durchführen zu können.

Im zweiten Block soll das rudimentäre Suchsystem erweitert werden um die Möglichkeit, Terme in der Suchanfrage semantisch zu annotieren. Dadurch soll ermöglicht werden, Homographen zu disambiguieren wie z.B. den Namen „Schneider“ von der Berufsbezeichnung „Schneider“.

Im dritten Block soll das Suchsystem um ein Subsystem zur Anfrageerweiterung erweitert werden, um den recall für bestimmte Suchen bestimmter Benutzergruppen zu erhöhen. In diesem Block soll das Suchsystem dann an eine gegebene Kollektion angepasst werden.

Nach Abschluss des dritten Blocks werden die Suchsysteme der Teilnehmer im Rahmen einer kleinen IR-Evaluation auf der gegebenen Kollektion gegeneinander getestet. Die Ergebnisse werden auf der Vorlesungs-Webseite veröffentlicht.

Die Aufgabenblätter der einzelnen Blöcke werden nach und nach zusammen mit evtl. notwendigen Dateien auf der Vorlesungs-Webseite[1] veröffentlicht.

1.3 Arbeitsweise

Die Bearbeitung der Aufgaben erfolgt in Einzelarbeit. Gruppen sind nicht zugelassen. Die Ergebnisse der einzelnen Blöcke werden auf unserer Webseite hochgeladen. Einzelheiten sind den jeweiligen Aufgabenblättern zu entnehmen.

Die Lösungen der einzelnen Blöcke sind als gezipptes Eclipse-Projekt abzugeben, das als UTF-8 kodiert ist. Jedes abgegebene Projekt muss in Eclipse Helios oder Indigo importierbar und von dort lauffähig sein. Nicht mit abgegeben werden sollen Dateien der Kollektion oder Indexe. Die Kollektions-Daten liegen bei uns vor und der Index wird von uns mithilfe Eures Projekt-Codes erstellt. Der Code ist daher

[1] `http://www.is.inf.uni-due.de/courses/ir_ws11/`

so zu gestalten, dass die Angabe der Verzeichnisse, in denen eine zu indexierende Kollektion liegt oder in das der Index geschrieben werden soll, konfigurierbar ist – entweder durch Änderung einer Konstante im Code oder über eine Properties-Datei.

Wenn abgegebener Code Syntax-Fehler enthält, die dazu führen, dass der Code nicht kompiliert werden kann, gilt der Block als nicht bestanden. Wenn abgegebener Code auf unseren Beispiel-Eingaben abstürzt oder unvorhergesehene bzw. fehlerhafte Ausgaben liefert, gilt der Block als nicht bestanden.

Im Praktikum soll Java 1.6 verwendet werden.

1.4 Kleiner Hinweis am Rande ;-)

In Java kann man Zeichenketten folgendermaßen konkatenieren (wir lesen hier aus einem `Reader` – z.B. per `FileReader`):

Listing 1: reader1.java

```
1  public String read(BufferedReader in) {
2      String out = "";
3      String read = null;
4      try {
5          while ((read = in.readLine()) != null) {
6              out += read; // hier wird konkateniert
7              out += "\n"; // hier auch
8          }
9      }
10     catch (IOException e) {
11     }
12     return out;
13 }
```

Dieses Verfahren ist sehr langsam, weil Java dabei sehr viele neue Objekte erzeugt und wieder wegwerfen muss. Was nämlich eigentlich passiert, ist das hier:

Listing 2: reader1.java

```
1  public String read(BufferedReader in) {
2      String out = "";
3      String read = null;
4      try {
5          while ((read = in.readLine()) != null) {
6                  StringBuilder o1 = new StringBuilder(out);
7                  o1.append(read);
8                  out = o1.toString();
9
10                 StringBuilder o2 = new StringBuilder(out);
11                 o2.append("\n");
12                 out = o2.toString();
13         }
14     }
15     catch (IOException e) {
16     }
17     return out;
18 }
```

Also für jedes `out += read` etc. wird ein neuer `StringBuilder` erzeugt, der neue String angehangen und das Ergebnis wieder ausgegeben. Der `StringBuilder` sowie der ursprüngliche String werden dann vom garbage collector entsorgt.

Du solltest das ruhig mal ausprobieren: es ist erstaunlich, wie langsam das tatsächlich ist.

Wesentlich (und man kann das gar nicht genug betonen) schneller ist folgender Code:

Listing 3: reader2.java

```
1  public String read(BufferedReader in) {
2      StringBuilder out = new StringBuilder();
3      String read = null;
4      try {
5          while ((read = in.readLine()) != null) {
6              out.append(read).append('\n');
7          }
8      }
9      catch (IOException e) {
10     }
11     return out.toString();
12 }
```

Evtl. kennst Du `StringBuffer` schon: der macht im Grunde dasselbe wie `StringBuilder`, aber ist für nebenläufigen Zugriff ausgelegt. Wenn es ausgeschlossen ist, dass zwei Threads auf denselben `StringBuffer` zugreifen, ist es ein bisschen schneller, `StringBuilder` zu verwenden. Das ist im Kontext des IR-Praktikums nicht so wichtig, aber bei Systemen, die entweder lange laufen oder von vielen Benutzern gleichzeitig verwendet werden, schon: Schneller bedeutet da mehr Benutzer pro CPU und Zeit und Wärme und damit auch Geld.