Algorithmische Anwendungen WS 05/06
Document Ranking

Ulrich Schulte (ai641@gm.fh-koeln.de)
Harald Wendel (ai647@gm.fh-koeln.de)

Inhaltsverzeichnis

1. Document Ranking

1.1 TF*IDF

Die Anzahl und thematische Vielfalt bestehender Informationsressourcen hat eine Dimension erreicht, die große Herausforderungen an entsprechende Werkzeuge zur Informationsbeschaffung stellt.

„The number of needles that can be found has increased, but so has the size of the haystack they are hidden in.“

Ein weit verbreiteter Ansatz der grundlegenden Techniken heutiger Suchsysteme ist das TF*IDF Schema. Dabei steht TF für *term frequency* und IDF für *inversed document frequency*.
Die Grundidee des TF-IDF-Schemas ist die Annahme, daß Schlüsselbegriffe, die in einem bestimmten Dokument relativ häufig, in der Grundgesamtheit der Dokumente aber relativ selten auftauchen, ein guter Indikator für den Inhalt eines Dokuments seien.

Formal ausgedrückt stellt sich das TF-IDF-Schema folgendermaßen dar:

$w_{ij} = tf_{ij} \cdot \log(\frac{N}{d_{fj}})$, wobei

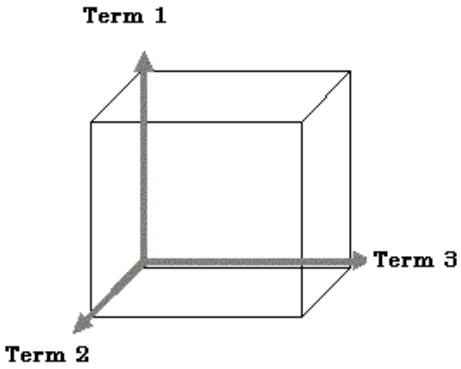
w_{ij} :Kombiniertes Gewicht des Wortes j in Dokument i

tf_{ij} :Absolute Häufigkeit des Wortes j in Dokument i

d_{fj} :Absolute Häufigkeit der Dokumente innerhalb der Gesamtheit von N Dokumenten, in denen Wort j mindestens einmal auftaucht

1.2 Das Vector Space Modell

Das Vector Space Modell ist ein algebraisches Modell, in dem Dokumente und Dokumentterme als Vektoren im mehrdimensionalen Raum dargestellt sind. Suchanfragen und Dokumente werden als Vektoren im Termraum projiziert.



Beispiel:

Wir gehen von einer Testkollektion mit den folgenden drei Dokumenten aus:

N_1 : "Shipment of gold damaged in a fire"
N_2 : "Delivery of silver arrived in a silver truck"
N_3 : "Shipment of gold arrived in a truck"

Die Ergebnisse der Anfrage „gold silver truck“ stellt folgende Tabelle dar:

		Anzahl, tf_i						**Gewichte,** $w_i = tf_i * IDF_i$			
Terme	**Q**	N_1	N_2	N_3	df_i	N/df_i	IDF_i	**Q**	N_1	N_2	N_3
a	0	1	1	1	3	3/3=1	0	0	0	0	0
arrived	0	0	1	1	2	3/2=1,5	0,1761	0	0	0,1761	0,1761
damaged	0	1	0	0	1	3/1=3	0,4771	0	0,4771	0	0
delivery	0	0	1	0	1	3/1=3	0,4771	0	0	0,4771	0
fire	0	1	0	0	1	3/1=3	0,4771	0	0,4771	0	0
gold	1	1	0	1	2	3/2=1,5	0,1761	0,1761	0,1761	0	0,1761
in	0	1	1	1	3	3/3=1	0	0	0	0	0
of	0	1	1	1	3	3/3=1	0	0	0	0	0
silver	1	0	2	0	2	3/2=1,5	0,4771	0,4771	0	0,9542	0
shipment	0	1	0	1	2	3/2=1,5	0,1761	0	0,1761	0	0,1761
truck	1	0	1	1	2	3/2=1,5	0,1761	0,1761	0	0,1761	0,1761

In den Spalten 1-5 konstruieren wir zunächst einen Index der Terme aus den Dokumenten und tragen jeweils die Anzahl der Terme in der Anfrage und die Anzahl der Terme innerhalb des jeweiligen Dokumentes ein.

In den Spalten 6-7 errechnen wir die IDF jedes Terms. In den letzten drei Spalten kann man dann die Gewichte der einzelnen Terme in den verschiedenen Dokumenten ablesen.

Ähnlichkeitsanalyse:
Wir betrachten Gewichte als Koordinaten im Vektorraum, in dem die Dokumente und die Anfrage als Vektoren repräsentiert sind. Um nun herauszufinden, welcher Dokument-Vektor näher am Anfrage-Vektor liegt, führen wir eine Ähnlichkeitsanalyse durch.

Per Definition hat ein Vektor einen Betrag und eine Richtung. Die Anfragevektoren beginnen alle im Nullpunkt und haben eine bestimmte Länge.
Da $a \cdot b = |(a)| \cdot |(b)| * \cos(\varphi)$, ist der Winkel φ ist ein Maß, wie sehr sich die Term-Vektoren unterscheiden. Damit haben wir mit dem $\cos(\varphi) = \frac{a \cdot b}{|(a)| \cdot |(b)|}$ ein Maß für die Similarität zwischen a und b.

Zunächst errechnen wir für jedes Dokument und Anfrage die Vektorlängen (Null-Terme werden ignoriert):

$$|N_1| = \sqrt{0{,}4771^2 + 0{,}4771^2 + 0{,}1761^2 + 0{,}1761^2} = \sqrt{0{,}5173} = 0{,}7192$$
$$|N_2| = \sqrt{0{,}1762^2 + 0{,}4771^2 + 0{,}9542^2 + 0{,}1761^2} = \sqrt{1{,}2001} = 1{,}0955$$
$$|N_3| = \sqrt{0{,}1761^2 + 0{,}1761^2 + 0{,}1761^2 + 0{,}1761^2} = \sqrt{0{,}1240} = 0{,}3522$$

$$|Q| = \sqrt{0{,}1761^2 + 0{,}4771^2 + 0{,}1761^2} = \sqrt{0{,}2896} = 0{,}5382$$

Danach berechnen wir die Skalarprodukte:

$$Q \cdot N_1 = 0{,}1761 \cdot 0{,}1761 = 0{,}0310$$
$$Q \cdot N_2 = 0{,}4771 \cdot 0{,}9542 + 0{,}1761 \cdot 0{,}1761 = 0{,}4862$$
$$Q \cdot N_3 = 0{,}1761 \cdot 0{,}1761 + 0{,}1761 \cdot 0{,}1761 = 0{,}0620$$

Nun berechnen wir die Ähnlichkeitswerte, indem wir die korrespondierenden Cosinus Werte berechnen:

$$\cos(N_1) = \frac{Q \cdot N_1}{|(Q)| \cdot |(N_1)|} = \frac{0{,}0310}{0{,}5382 \cdot 0{,}7192} = 0{,}0801$$
$$\cos(N_2) = \frac{Q \cdot N_2}{|(Q)| \cdot |(N_2)|} = \frac{0{,}4862}{0{,}5382 \cdot 1{,}0955} = 0{,}8264$$
$$\cos(N_3) = \frac{Q \cdot N_3}{|(Q)| \cdot |(N_3)|} = \frac{0{,}0620}{0{,}5382 \cdot 0{,}3522} = 0{,}3271$$

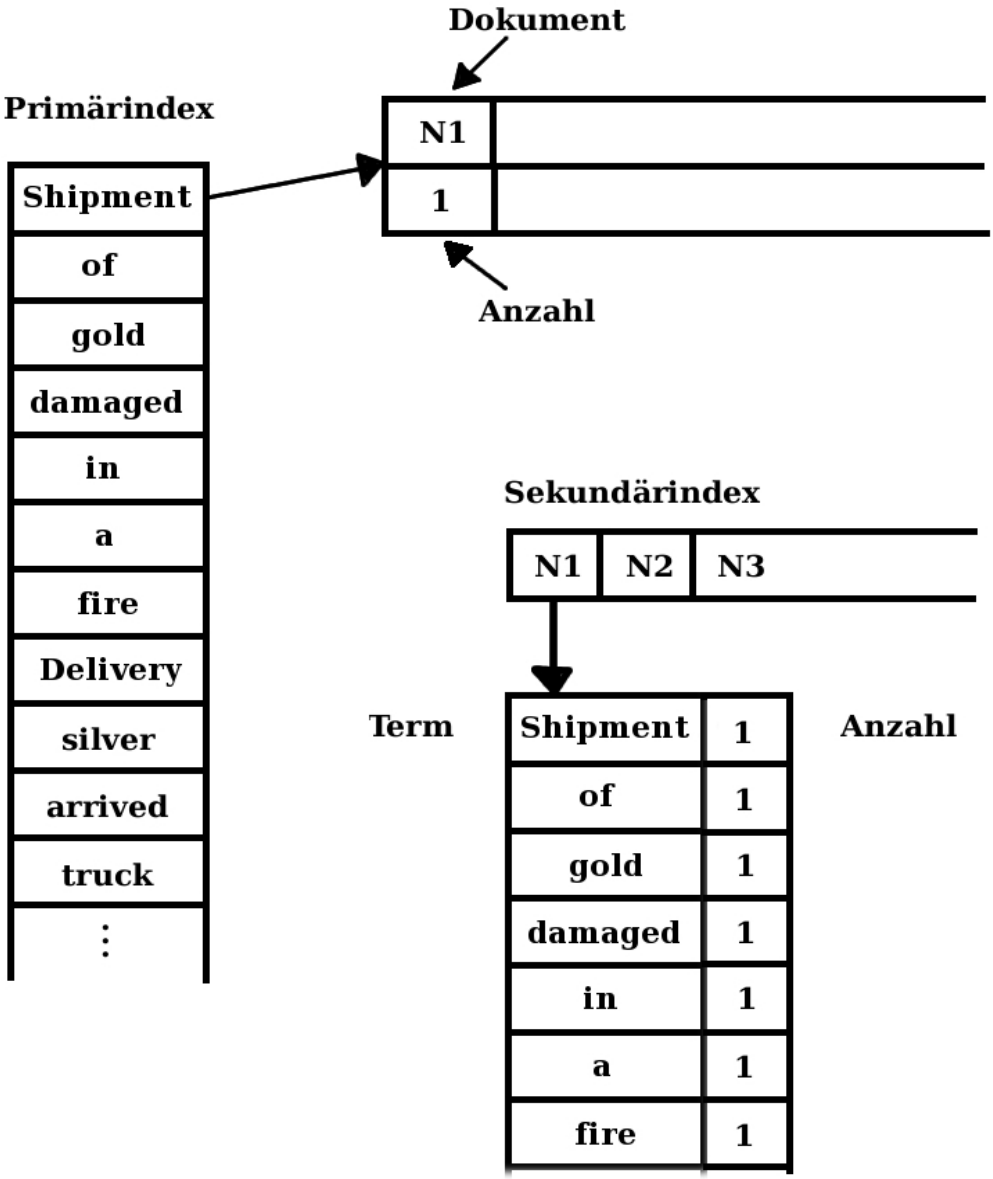
Zum Schluß ordnen wir unsere Dokumente absteigend gemäß Ihrer Ähnlichkeitswerte:

Rang 1: Dokument2 = 0,8246

Rang 2: Dokument3 = 0,3271

Rang 3: Dokument1 = 0,0801

Der invertierte Index:



Im Haupt- oder Primärindex werden Terme indiziert, um die Dokumente schnell zu finden. Der Index wird also nach den Termen sortiert und verweist auf die jeweiligen Dokumente und ihre Häufigkeit innerhalb diesem.

Für das Scoring hingegen ist der Sekundärindex für die Termfrequenz von Bedeutung. In ihm verweist ein bestimmtes Dokument auf seine Terme und ihre Häufigkeit innerhalb des Dokumentes.

2. Das Vektor Modell mit normalisierten Frequenzen

Da Terme, die oft in einem Dokument auftauchen, ein höheres Gewicht erhalten, als Terme, die weniger oft wiederholt werden, ist das Modell bisher noch anfällig für „keyword spamming“. Im Folgenden wollen wir versuchen die Frequenzen der Terme zu normalisieren.

2.1 Normalisierte Dokument-Frequenz

Die normalisierte Frequenz eines Terms i in Dokument j ist gegeben mit:

$f_{i,j} = tf_{i,j} / max\, tf_{i,f}$, wobei

$f_{i,j}$	die normalisierte Frequenz,
$tf_{i,j}$	die Frequenz von Term i in Dokument j und
$max\, tf_{i,f}$	die maximale Frequenz von Term i in Dokument j ist.

Betrachten wir zum Beispiel ein Dokument mit folgen Begriffsanzahlen:

major, 1

league, 2

baseball, 4

playoffs, 5

Da *playoffs* am meisten vorkommt, errechnen sich die normalisierten Frequenzen zu:

major, 1/5 = 0.20

league, 2/5 = 0.40

baseball, 4/5 = 0.80

playoffs, 5/5 = 1

2.2 Normalisierte Anfrage Frequenzen

Die normalisierte Frequenz eines Terms i in einer Anfrage Q ist gegeben mit:

$f_{Q,i} = 0{,}5 + 0{,}5 \cdot tf_{Q,i} / max\, tf_{Q,i}$, wobei

$f_{Q,i}$	die normalisierte Frequenz,
$tf_{Q,i}$	die Frequenz von Term i in Anfrage j,
$max\, tf_{Q,i}$	die maximale Frequenz von Term i in der Anfrage j.

Für die Anfrage *Q = major major league* ergeben sich folgende Frequenzen:

major, 2

league, 1

da major zweimal in der Anfrage vorkommt, sind die normalisierten Frequenzen:

major, (0.5 + 0.5*2/2) = 1

league, (0.5 + 0.5*1/2) = 0.75

2.3 Normalisiertes Gewicht

Unter Berücksichtigung der normalisierten Frequenzen, können wir das Gewicht des Terms i in Dokument j folgendermaßen beschreiben:

$$w_{i,j} = \frac{tf_{i,j}}{max\, tf_{i,j}} \cdot \log\left(\frac{N}{df_i}\right)$$

Und das Gewicht von Term i in der Anfrage Q kann beschrieben werden als:

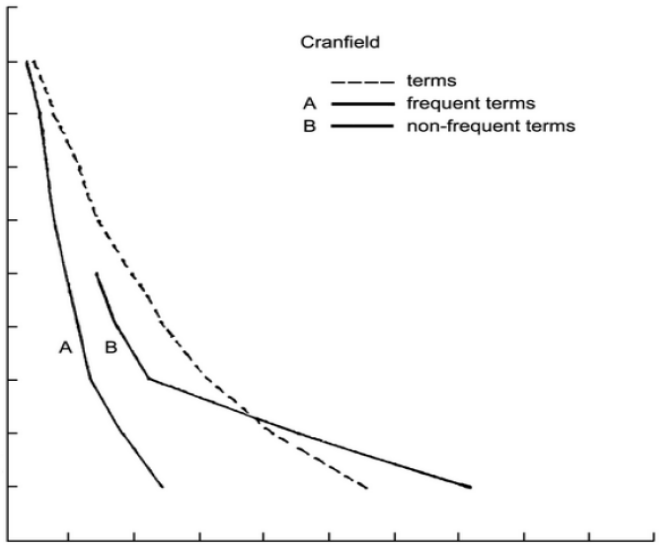
$$w_{Q,i} = \left(0{,}5 + 0{,}5 \cdot \frac{tf_{i,j}}{max\, tf_{i,j}}\right) \cdot \log\left(\frac{N}{df_i}\right)$$

3. Literaturrechereche „Exhaustivity and Specificity“

Während unserer Recherche stießen wir immer wieder auf den Namen Karen Sparck Jones und ihre Versuche mit Ranking Algorithmen an Testkollektionen.
In dem Originalartikel geht Jones vor allem auf die zwei Begriffe Exhaustivity und Specificity ein. „Exhaustivity“ ist dabei die Anzahl der Terme, die eine Dokumentbeschreibung enthält und die „Specificity“ eines Terms ist die Anzahl der Dokumente, die es enthalten, also die Größe des korrespondierenden Dokumentenvektors.

Oft benutzte Terme fungieren im Ranking daher als eher unspezifische Terme, auch wenn ihre Bedeutung im eigentlichen Sinne vielleicht sehr spezifisch ist. Man kann sich also vorstellen, dass ein Term wie etwa „Getränk“ beim Ranking wesentlich unspezifischer ist und einen größeren Dokumentenvektor hat, als Terme wie „Kaffee“, „Tee“ oder „Kakao“.

Jones erster naiver Ansatz beim Ranking eine höhere Relevanz der Ergebnisse zu erreichen, war es, einfach die ersten 25 häufigsten Terme aus den jeweiligen Testkollektionen zu löschen. Dieser Ansatz führte jedoch zu einer wesentlich schlechteren Relevanz der Ergebnisse. Betrachtet man die einzelnen „precision-recall-Graphen“ der Testkollektionen, so wird auch klar warum.



Zeichnung 1: precision-recall-Graph der Cranfield Testkollektion

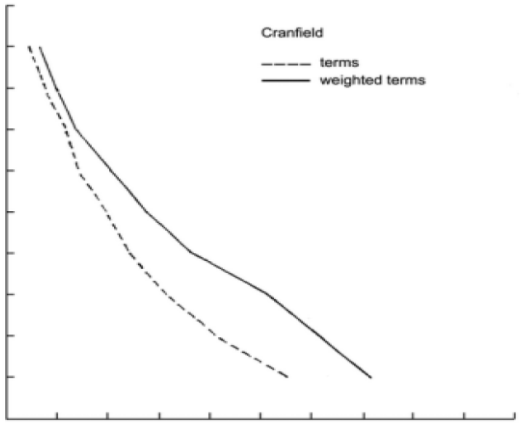
An der y-Achse ist der *recall*, also die Anzahl der Treffer abzulesen und auf der x-Achse die

precision, also die Relevanz der jeweiligen Treffer. Betrachtet man nun getrennt A (frequente Terme) und B (nicht-frequente Terme), so fällt auf, daß vor allem im mittleren *recall* die frequenten Terme deutlich zu einer höheren Relevanz beitragen.

Das Entfernen hochfrequenter Terme verschlechtert also im Durchschnitt die Qualität des Ergebnisses.
Ein weiterer und einleuchtender Ansatz war nun die Einführung einer Gewichtsfunktion. Die einzelnen Terme werden durch sie entsprechend ihrer Frequenz innerhalb der Kollektion verschieden gewichtet:

$weight(term) = f(N) - f(n) + 1$, dabei ist N die Anzahl der Dokumente
n die Termfrequenz

Bei kunjunktiven Anfragen werden die Gewichte der Terme einfach addiert.
Die Anwendung der Gewichtsfunktion führte zu einer deutlichen Verbesserung bei allen Testkollektionen.



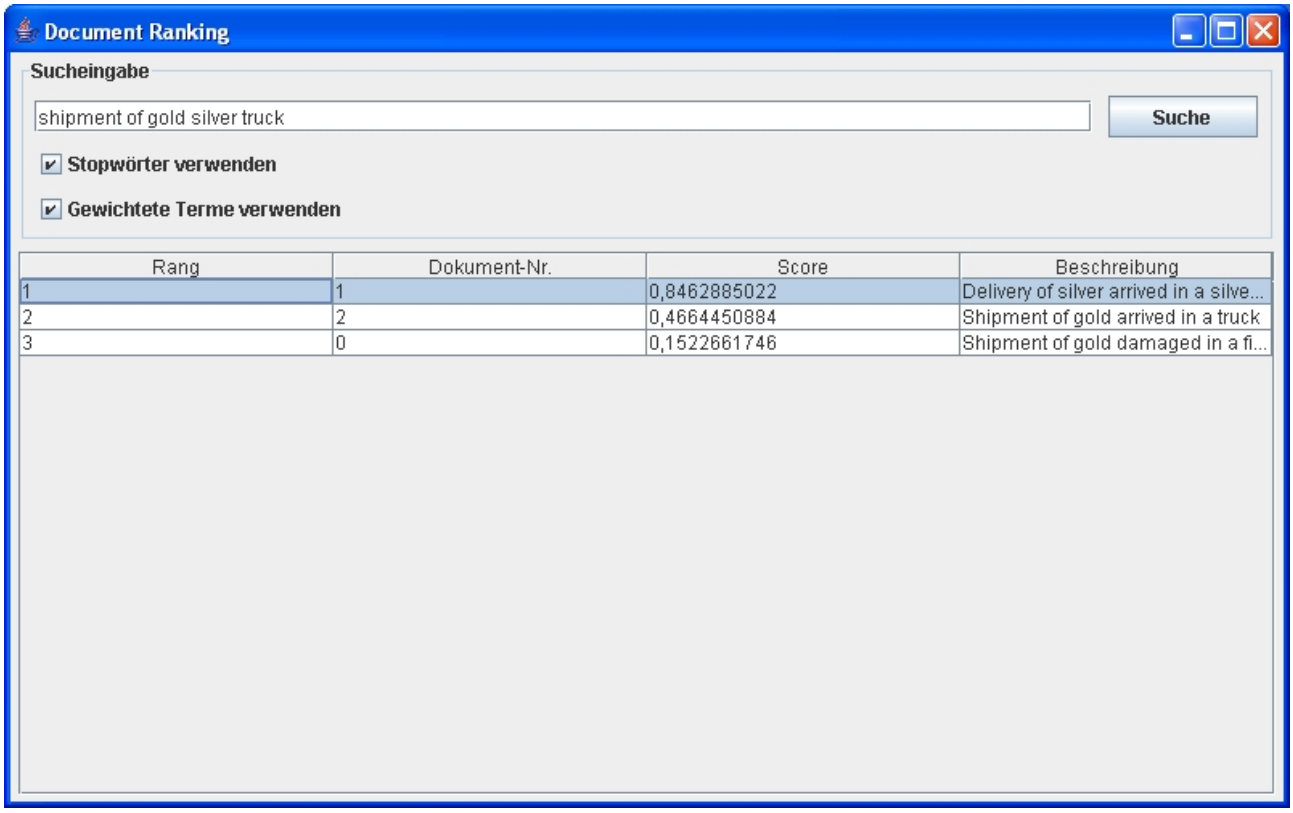
Zeichnung 2: precision-recall-Graph der Cranfield Testkollektion

4. Implementierung

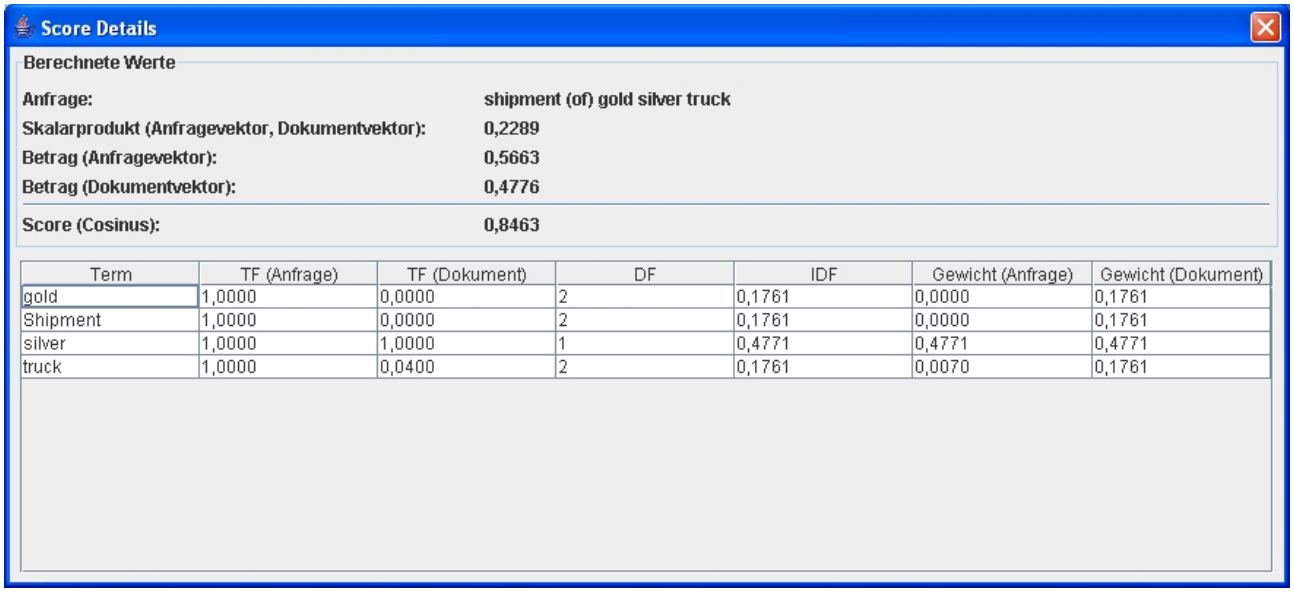
Das Ergebnis unserer Implementierung ist eine Suchmaschine in JAVA, die eine kleine Dummy-Datenbank mit Dokumenten einliest und indiziert. Im Eingabefeld kann eine Sucheingabe eingegeben werden.

Durch Anklicken der Checkboxen kann man das Verwenden von gewichteten Termen oder Stopwörtern veranlassen. Die Nutzung von Stopwörtern ist ein Verfahren, welches hauptsächlich zur Reduktion der Indexgröße dient. Die Idee hierbei ist es, Wörter welche sehr häufig vorkommen und für Suchen meist irrelevant sind, gar nicht erst im Index abzulegen. Diese Wörter umfassen meist alle Arten von Füllwörtern wie Artikel (der, eine), Adverben (hier, da), Präpositionen (in, auf), Pronomen (ich, du) und Konjunktionen (und, oder).

Nachdem eine Suchanfrage gestartet wurde, erscheint eine Tabelle mit dem Ranking der betreffenden Dokumente.



Durch Doppelklicken auf eines der Rankingergebnisse öffnet sich ein Popup, in dem alle zum Scoring berechneten Werte aufgelistet sind.



Term	TF (Anfrage)	TF (Dokument)	DF	IDF	Gewicht (Anfrage)	Gewicht (Dokument)
gold	1,0000	0,0000	2	0,1761	0,0000	0,1761
Shipment	1,0000	0,0000	2	0,1761	0,0000	0,1761
silver	1,0000	1,0000	1	0,4771	0,4771	0,4771
truck	1,0000	0,0400	2	0,1761	0,0070	0,1761

Der Quellcode zu den beiden wichtigsten Methoden *search* und *score* findet sich im Anhang, sowie zahlreiche Kommentare und Erläuterungen zu diesem.

5. Anhang

```
/**
 * Führt eine Volltextsuche durch
 *
 * @param query         Anfrage
 * @param a             Stoppwort-Analysierer
 * @return              sortierte Menge von Treffern
 */
public SortedSet<Hit> search(Query query, Analyzer a) {

      /*
       * Ausführen der Anfrage:
       *
       *          - Anfragestring wird in Token zerlegt
       *
       *          - mit Hilfe des Analyzers wird überprüft,
       *            ob es sich um ein Stoppwort handelt
       *
       *          - wenn nicht wird aus dem Index eine Referenz
       *            auf den Term auf den Term geholt
       *
       *                      Term t = index.get(token);
       *
       *          - wenn der Term im Index vorhanden ist (t != null)
       *            dann wird der Term zur Liste der Anfrageterme
       *            hinzugefügt
       */
      query.execute(this, a);

      /*
       * Menge aller Dokumente, die zu der Anfrage passen, bzw.
       * mindestens einen Term der Anfrage enthalten
       */
      SortedSet<Integer> retrievedDocs =
                           new TreeSet<Integer>(new IntComparator());

      /*
       * für alle Terme der Anfrage
       */
      for (Term t : query.terms()) {
            /*
             * für alle Dokumente die den Term enthalten
             */
            for (int docId : t.getDocs()) {
                  /*
                   * Wenn das Dokument noch nicht in der Menge der
                   * gefundenen Dokumente entahlten ist, dann wird
                   * es hinzugefügt
                   */
                  if (!retrievedDocs.contains(docId)) {
                        retrievedDocs.add(docId);
                  }
            }
      }
```

```
	/*
	 * Menge aller Treffer, geordnet nach ihrem Rang
	 */
	SortedSet<Hit> hitSet = new TreeSet<Hit>(new HitComparator());

	/*
	 * für alle gefundenen Dokumente
	 */
	for (int doc : retrievedDocs) {
		/*
		 * wird der Rang des Dokuments berechnet und ein neuer
		 * Treffer erzeugt
		 */
		hitSet.add(new Hit(doc, scorer.score(this, doc, query)));
	}

	return hitSet;
}
```

```
public float score(Index index, int docId, Query query) {

	/*
	 * dlen: Betrag des Dokumentvektors
	 * qlen: Betrag des Anfragevektors
	 */
	float dlen = 0.0f, qlen = 0.0f;

	/*
	 * Anzahl der Dokumente im Index
	 */
	float docCount = index.docCount();

	/*
	 * zu bewertendes Dokument
	 */
	Document doc = index.getDoc(docId);

	/*
	 * dWeights: Vektor der Gewichte der Terme im Dokument
	 * qWeights: Vektor der Gewichte der Terme in der Anfrage
	 */
	ArrayList<Float> qWeights = new ArrayList<Float>();
	ArrayList<Float> dWeights = new ArrayList<Float>();

	/*
	 * für alle Terme im Dokument
	 */
	for (Term t : doc.terms()) {
		/*
		 * berechne das Gewicht aus:
		 *
		 *    t.tf(docId) - Frequenz des Terms im Dokument
		 *    docCount    - Anzahl der Dokumente im Index
		 *    t.df()      - Anzahl der Dokument die den Term enthalten
		 */
		float dw = weight(t.tf(docId), docCount, t.df());
		/*
		 * füge das Quadrat des Gewichts zum Betrag
		 * des Vektors hinzu
		 */
		dlen += dw * dw;
	}

	/*
	 * für alle Terme in der Anfrage
	 */
	for (Term t : query.terms()) {
		/*
		 * berechne das Gewicht
		 * (analog zur Berechnung des Gewichts im
		 *  Dokumentenvektor)
		 */
		float dw = weight(t.tf(docId), docCount, t.df());
		float qw = weight(query.tf(t), docCount, t.df());
		qlen += qw * qw;

		dWeights.add(qw);
		qWeights.add(dw);
```

```
        }

        /*
         * Die Beträge der Vektoren
         */
        dlen = (float)Math.sqrt(dlen);
        qlen = (float)Math.sqrt(qlen);

        /*
         * Das Skalarprodukt der beiden Vektoren
         */
        float dotp = dot(qWeights, dWeights);

        /*
         * Bewertung:
         *
         *    Skalarprodukt / (Länge Dokumentvektor * Länge Anfragevektor)
         */
        return dotp / (dlen * qlen);
}
```

6. Quellen

- Journal of Documentation, Volume 28 Number 1 1972 pp. 11-21; „Exhaustivity and specificity“ von Karen Spärck Jones, Computer Laboratory, University of Cambridge, Cambridge, UK
- http://www.miislita.com/term-vector/term-vector-2.html
- http://www.miislita.com/term-vector/term-vector-3.html
- http://www.miislita.com/term-vector/term-vector-4.html
- http://de.wikipedia.org/wiki/Vektorraum-Retrieval
- http://www.greenbuilt-research.com/search-technology/archive/ir-vector.html