Information Retrieval

Seminararbeit

des Studienganges Angewandte Informatik
an der Dualen Hochschule Baden-Württemberg Mannheim

von

Jesse-Jermaine Richter, Jonas Seng

29.04.2019

Matrikelnummer, Kurs:	8787549/1980179, TINF16AIBI
Ausbildungsfirma:	DZ BANK AG, Frankfurt
Betreuer der Ausarbeitung:	Herr Prof. Dr. Karl Stroetmann

Erklärung

Wir versichern hiermit, dass wir unsere Seminararbeit mit dem Thema: „Information Retrieval“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum	Unterschrift
Ort, Datum	Unterschrift

In dieser Seminararbeit wird das Thema „Information Retrieval“ anhand einer lokalen Suchmaschine näher erläutert...

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungstabelle

Abkürzung:	**Bedeutung:**
IR	Information Retrieval
I/O	Input/Output

1 Einleitung

1.1 Was ist Information-Retrieval?

Information-Retrieval (IR) beschreibt das Bereitstellen spezieller Informationen aus einer großen und unsortierten Datenmengen. Dieses Themengebiet fällt unter Informatik, Informationswissenschaften sowie Computerlinguistik und ist ein wesentlicher Bestandteil von Suchmaschinen wie zum Beispiel Google.
Das Thema besitzt bereits seit einigen Jahren eine hohe, aber dennoch steigende Relevanz. Die Gründe der hohen Relevanz von IR liegen vor allem beim Einsatz von Suchmaschinen. Diese sind in Zeiten des Internets die wohl wichtigste Form der Informationsbeschaffung. Aufgrund der immer schneller steigenden Informationsmengen wird das Thema künftig weiter an Relevanz gewinnen. Unternehmen und Privatanwendern wird eine immer wachsende Menge von Informationen zugänglich, die organisiert werden muss, damit relevante bzw. spezifisch gesuchte Informationen jederzeit und ohne Verzögerung gefunden werden.
Um das Ziel der Bereitstellung von Informationen gewährleisten zu können, wird erst eine Durchsuchung und Gewichtung sämtlicher Informationen bzw. Dokumente, die später gefunden werden sollen, durchgeführt. Das zentrale Objekt der Informationsrückgewinnung stellt der invertierte Index dar, dessen Aufbau und Funktionsweise in den nächsten Kapiteln ausführlich erläutert wird. Im Verlauf dieser Arbeit wird die Komprimierung des Indexes sowie das TF-IDF-Maß, welches zur Beurteilung der Relevanz eines Dokumentes genutzt wird, im Fokus stehen.
Die theoretischen Hintergründe des invertierten Index, der Komprimierung und des TF-IDF-Maßes werden durch eine Beispiel-Implementierung einer lokalen Suchmaschine in der Programmiersprache Python veranschaulicht.

1.2 Ziel der Arbeit

Ziel der Arbeit ist es, ein grundlegendes Verständnis des Themenkomplexes Information-Retrieval zu vermitteln. Das umfasst auch die theoretischen Hintergründe, die für die später vorgestellten Beispiel-Implementierungen notwendig sind.
Die Beispiel-Implementierung soll hauptsächlich die folgenden Themengebiete umfassen:

- Aufbau eines invertierten Indexes
- Approximierende Beurteilung der Relevanz eines gefundenen Dokuments mittels TF-IDF
- Komprimierung des invertierten Indexes

Die in dieser Arbeit vorgestellte Implementierung hat nicht den Anspruch auf eine hohe Performance, vielmehr dient diese dem Zwecke der praxisnahen Veranschaulichung der Funktionsweise von IR-Systemen.

1.3 Stand der Forschung

Dieser Abschnitt umreißt kurz den aktuellen Stand der Forschung. Dazu werden zwei Modelle von Information Retrieval knapp beschrieben, die für die Entwicklung einer lokalen Suchmaschine, von Bedeutung sind. Es wird jedoch nur auf ein Modell im Verlauf dieser Arbeit eingegangen.

1.3.1 Vector Space Model

Das Vector Space Model, zu deutsch Vektorraummodell, repräsentiert Dokumente und Anfragen als hochdimensionale, metrische Vektoren [6]. Der Anfrage-Vektor wird beim Retrieval-Prozess mit den Dokumenten-Vektoren verglichen. Dabei werden jedoch nur Dokumente betrachtet, welche mit der Anfrage in Verbindung stehen [10]. Welche Dokumente betrachtet werden, wird mithilfe des invertierten Index ermittelt.

Es gibt verschiedene Maße, mit denen die Vektoren miteinander verglichen werden. Der einfachste Ansatz besteht darin, den Abstand zwischen Anfrage-Vektor und Dokumenten-Vektor zu berechnen, jedoch ist dies kein sehr gutes Maß. Besser und weit verbreitet ist deshalb das Cosinus-Maß (oder Cosinus-Ähnlichkeit), welches den Winkel zwischen Anfrage-Vektor und Dokumenten-Vektor angibt. Je kleiner der Winkel zwischen den Vektoren, desto höher ist die Relevanz des Dokuments [2].
Dieses Modell wir im Rahmen dieser Arbeit genauer beleuchtet und als Grundlage für die Beispiel-Implementierung genutzt.

2 Information Retrieval - Theoretische Grundlagen

2.1 Problemstellung

Im Rahmen dieser Arbeit soll beschrieben werden, wie mithilfe von „Information Retrieval“ Informationen aus einer großen, unsortierten Datenmenge - nach Relevanz sortiert - bereitgestellt werden kann. Dabei bekommt das System eine vom Nutzer gestellte Abfrage, auch Query genannt, und versucht auf deren Basis, Daten, die meist als Dokumente vorliegen, zurückzuliefern. Im Gegensatz zu Abfragen im Datenbankumfeld beinhaltet die Query jedoch unzureichende Informationen, um ein spezielles Element eindeutig identifizieren zu können. Dies soll ein IR-System auch nicht leisten. Vielmehr sollen Ergebnisse zurückgeliefert werden, die mit hoher Wahrscheinlichkeit Relevanz bzgl. der gestellten Query besitzen. Der Nutzer selektiert dann die für diesen nötigen Dokumente.
Mathematisch lässt sich dies folgendermaßen formulieren: Aus einer Dokumentenmenge D soll mithilfe einer Funktion eine Teilmenge D_1 von D ermittelt werden, die relevant für eine Abfrage q ist.
Um diese Funktion sinnvoll definieren zu können, muss jedoch zuvor die Menge aller Queries, sowie die Menge aller Tokens definiert werden. Zuvor sollen jedoch die Vokabeln „Token“, „Typ“ und „Term“ geklärt werden:

Definition 2.1 (Token) *Unter einem Token wird eine zusammenhängende Zeichenkette verstanden, die innerhalb eines Dokuments vorkommt [4].*

Tokens werden häufiger auch als „Wörter“ bezeichnet, dies wird auch im Rahmen dieser Arbeit so gehandhabt.

Definition 2.2 (Typ) *Ein Typ bezeichnet eine Klasse von Tokens, die dieselben Zeichen in derselben Reihenfolge enthalten [4].*

Definition 2.3 (Term) *Ein Term ist ein Typ, welcher im Dictionary eines IR-Systems vorkommt [4].*

Ein Beispiel, wie Tokens, Typen und Terme aussehen können, folgt in 2.3.2.

Definition 2.4 (Menge aller Terme) *Sei $d \in D$ ein Dokument. Die Menge T_d ist nun die Menge aller Wörter, die in dem Dokument d enthalten sind: $T_d = \{t_1, .., t_n\}$.*
Die Menge T ist die Menge aller Terme, die in den Dokumenten aus D vorkommen, also: $T = T_{d_1} \cup .. \cup T_{d_i}$ mit $i \in N$ und $d_i \in D$.

Mithilfe der Definition 2.4 kann nun die Menge aller möglichen Queries definiert werden:

Definition 2.5 (Menge aller möglichen Queries) $Q \subseteq 2^T$

Definition 2.6 (Retrievalfunktion) *Eine Funktion $f: Q \to 2^D$ heißt Retrievalfunktion, wobei Q die Menge aller Queries ist.*

Nachdem die Problemstellung formuliert ist, muss eine Strategie entwickelt werden, wie die Retrievalfunktion nach Definition 2.6 dargestellt bzw. umgesetzt wird.

2.2 Strategiefindung

Dieser Abschnitt bietet eine Übersicht, wie das (im Folgenden vorgestellte) IR-System arbeiten soll.
Als Vorarbeit werden alle Dokumente, die im Index aufgenommen werden, in eine Codierung wie ASCII oder Unicode umgewandelt. Dazu wird ein Tool genutzt, das hier nicht weiter von Relevanz sein wird. Es sollen mindestens all diejenigen Dokumente in den Index aufgenommen werden, die im PDF-Format vorliegen.
Der erste Schritt, der das IR-System an sich leisten muss, ist das Erstellen von Tokens (siehe Definition 2.1). Dazu wird jedes Dokument in Tokens aufgespalten. Ein Token ist in den meisten Fällen ein Wort. Satzzeichen wie Leerzeichen, Kommata usw. sollen nicht

als Tokens behandelt und dementsprechend ignoriert werden.
Für jedes Token wird es später im Index einen Eintrag geben, der eine Liste mit weiteren Informationen hält. Diese Liste muss mindestens die Dokument-IDs speichern, in welchen das Token steht. In diesen Listen werden häufig noch weitere Informationen hinterlegt, beispielsweise die Häufigkeit eines Tokens.
Der zweite Schritt besteht darin, einen Algorithmus zu entwerfen, der eine Query entgegennimmt und auf Basis der Query und des Index eine Liste von relevanten Dokumenten ausgibt. Dieser Algorithmus wird das in der Einleitung kurz vorgestellte Vektorraummodell verwenden. Weiter wird dieser für die Ermittlung der Relevanz die sogenannte TF-IDF-Gewichtung nutzen. Diese wird später noch ausführlich vorgestellt.
Neben diesen beiden Punkten wird der Index komprimiert, um Speicherplatz zu sparen (und die Performance zu erhöhen).

2.3 Tokenization

2.3.1 Vorarbeiten

Bevor aus Dokumenten Tokens erzeugt werden können, müssen einige Fragen beantwortet werden. Eine Frage ist, welche Dokumente betrachtet werden sollen und wie man ein Dokument definiert. Ein Beispiel soll das Problem veranschaulichen:
Angenommen das IR-System soll dazu dienen Dokumente auf der Festplatte eines Computers zu finden. In diesem Szenario kann jede in einem Ordner gelistete Datei als Dokument angesehen werden. Dies wäre der einfachste Fall. Jedoch ist dies meist nicht erwünscht. So sollen beispielsweise bestimmte Dateitypen von der Suche ausgeschlossen werden. In UNIX existiert ein Dateityp, welcher mehrere Mails pro Datei speichert. Hier wird jede Mail als einzelnes Dokument angesehen. Daraus folgt, dass die Maildatei in mehrere Dokumente aufgespalten werden muss [4]. Umgekehrt gibt es Szenarien, in denen mehrere Dokumente zu einem Dokument zusammengefasst werden müssen, um bei der Suche nutzbare Ergebnisse zu erzielen [4]. Auch hier dienen E-Mails wieder als Beispiel: In vielen Mails werden Anhänge versandt. Häufig sind dies pdf-Dateien und Dateien von Text-Bearbeitungsprogrammen. Die Anhänge sollen zu dem E-Mail-Dokument gezählt werden,

an dem sie angehängt wurden.
Ein weiteres Problem, das gelöst werden muss, um die Dokumente verarbeiten zu können, ist die Codierung der Inhalte der Dokumente. Hierbei müssen Dokumente, die meist in vielen unterschiedlichen Codierungen vorliegen, zu einer definierten Codierung überführt werden [4].
Diese Probleme der „Vorarbeit“ werden in der später gezeigten Beispielimplementierung nicht behandelt, dies wird von anderen Tools übernommen.

2.3.2 Tokenerzeugung

Sobald geklärt ist, in welcher einheitlichen Codierung die Dokumente vorliegen und was als Dokument, im Englischen auch „document unit“ genannt, verstanden wird, kann ein Dokument in Tokens aufgeteilt werden.
Eine wichtige Frage, die im Rahmen der Tokenerzeugung geklärt werden muss, ist, welche Zeichenketten als Token behandelt werden. Kommata, Punkte und sonstige Satzzeichen haben keine sinnvolle Bedeutung im Zusammenhang mit Information Retrieval. Diese Zeichen können somit aus Tokens entfernt bzw. während der Tokenerzeugung überlesen werden [4]. Nach der Definition 2.1, erzeugt der Text

Beispielsatz, der ein Komma und ein Punkt hat.

die Tokens:

$$[\textit{Beispielsatz}, \textit{der}, \textit{ein}, \textit{Komma}, \textit{und}, \textit{ein}, \textit{Punkt}, \textit{hat}]$$

Die dazugehörige Menge von Typen sieht nach Definition 2.2 so aus:

$$Types = \{\textit{Beispielsatz}, \textit{der}, \textit{ein}, \textit{Komma}, \textit{und}, \textit{Punkt}, \textit{hat}\}$$

Die erzeugte Menge von Termen könnte nach Definition 2.3 - je nach Verarbeitung der Typen - wie folgt aussehen:

$$Terms = \{\textit{beispielsatz}, \textit{der}, \textit{ein}, \textit{komma}, \textit{und}, \textit{punkt}, \textit{hat}\}$$

Einige Information Retrieval-Systeme nutzen darüber hinaus sogenannte „stop words“. Das sind Wörter, die in sehr vielen Dokumenten in großer Anzahl vorkommen und damit wenig Bedeutung für die Suche besitzen [4]. Beispiele für solche Wörter sind „ist“, „sein“

und „und“. Jedoch funktioniert diese Technik später beim Suchen schlechter als zunächst angenommen. Das Wort „sein“ kann beispielsweise als Verb oder als Pronomen in einem Dokument vorkommen. Als Pronomen kann dieses Wort durchaus wichtig sein für eine Suche (beispielsweise innerhalb eines Buchtitels), wird jedoch als stop word aussortiert. Daher werden in neuen IR-Systemen entweder gar keine stop words oder nur eine geringe Anzahl stop words genutzt.

Eine weitere Möglichkeit solche Wörter zu filtern, ist „Stemming“. Diese Methode führt Wörter auf ihren Wortstamm zurück [4] [3]. Dadurch wird die Anzahl der Terme, die im Index gespeichert werden, stark gesenkt. Allerdings bringt diese Methode eine Unschärfe mit sich. Damit ist gemeint, dass zwei nicht verwandte Wörter auf denselben Wortstamm zurückgeführt werden, wodurch bei der späteren Suche nach einem der beiden Urpsrungswörter auch Ergebnisse zurückgeliefert werden, die irrelevant für die Query sind [4].

Weiter existieren sprachspezifische Probleme. Beispielsweise wird im Englischen häufig mit Kurzformen von Wörtern gearbeitet, so wird „are not“ zu „aren't“. Es muss geklärt werden wie mit solchen Formen umgegangen werden soll. Ein Ansatz ist Query Preprocessing. Hierbei werden Wörter dieser Art, die in der Query und in den Dokumenten stehen, in eine einheitliche Form gebracht [4]. Darüber hinaus gilt es zu beachten, dass Eigennamen wie „Hewlett-Packard“ nicht oder nur nach bestimmten Regeln prozessiert werden dürfen. Welche Wörter als Eigenname behandelt werden und welche nicht, kann mittels Machine Learning-Verfahren oder auf Basis eines großen Vokabulars gelöst werden.

3 Invertierter Index

Dieser Abschnitt wird die Idee des invertierten Index vorstellen sowie die Erzeugung des invertierten Index erläutern. Darüber hinaus wird aufgezeigt, wie die Verarbeitung einer Query mittels des invertierten Index funktioniert. Ab jetzt werden die Ausdrücke „invertierter Index“ und „Index“ synonym verwendet.

3.1 Grundlegender Aufbau

Der invertierte Index kann als ein Dictionary betrachtet werden, welches für jeden Term, der in der Dokumentenmenge vorkommt, einen Eintrag hält [4] [3]. Dieser Eintrag wiederum ist eine Liste. Diese hält mindestens eine eindeutige Dokumenten-ID, meist jedoch darüber hinaus weitere Informationen. Beispiele für solche weiteren Informationen sind Häufigkeit, in der ein Term in einem Dokument vorkommt, die Position und die umliegenden Wörter in der Nähe zum Term t [4]. Die Listen, die zu jedem Term angelegt werden, heißen Posting-Listen [4].
Den invertierten Index kann man folgendermaßen visualisieren:

Nummer	Term	Dokumente
1	als	5
2	auch	3
3	das	1, 2, 6
4	der	1, 2, 5
5	dritter	3
6	ein	3
7	ende	6
8	er	4
9	erste	1, 5
10	folgt	3
11	hier	2, 6
12	ist	1, 2, 6
13	länger	4
14	satz	1, 3
15	und	2, 5, 6
16	wobei	4
17	zweite	2, 5

Abbildung 3.1: Beispiel für einen invertierten Index [1]

3.2 Umsetzung eines invertierten Index

Nachdem das Prinzip des invertierten Index klar ist, steht die Frage im Raum wie dieser umgesetzt werden kann. Es liegt nahe als Datenstruktur einen Trie einzusetzen. Ein Trie ist ein spezieller Suchbaum, welcher besonders gut zum Suchen von Zeichenketten geeignet ist [4].
Alternativ kann über den Einsatz einer Hashmap nachgedacht werden, jedoch führt dies zu folgendem Problem: Angenommen die Terme einer vom Nutzer eingegebenen Query sind nicht in der Hashmap vorhanden, dann wird die Hash-Funktion keinen passenden Eintrag finden. Da in einer Hashmap Wörter, die ähnlich zueinander sind, nicht unbedingt benachbart gespeichert werden und keinerlei Information darüber bekannt ist, wo zur Query ähnliche Wörter gespeichert sind, kann im Falle, dass ein oder mehrere Wörter der Query nicht vorhanden sind, nicht mit geringem Aufwand nach ähnlichen Wörtern gesucht werden. Bei Tries besteht dieses Problem nicht [4].

Da die Beispiel-Implementierung, die später vorgestellt wird, jedoch möglichst einfach sein soll, wird zunächst eine Hashmap als Datenstruktur verwendet.

3.2.1 Tries

Ein Trie wird auf der Basis einer Menge von Zeichenketten aufgebaut. Jede Zeichenkette, die gefunden werden muss, ist innerhalb des Tries repräsentiert.
Erreicht wird dies dadurch, dass ein Knoten jeweils ein Zeichen repräsentiert und eine Liste mit Verweisen auf die nächsten möglichen Knoten hält, basierend auf einem weiteren Zeichen. Die folgende Abbildung zeigt einen Trie.

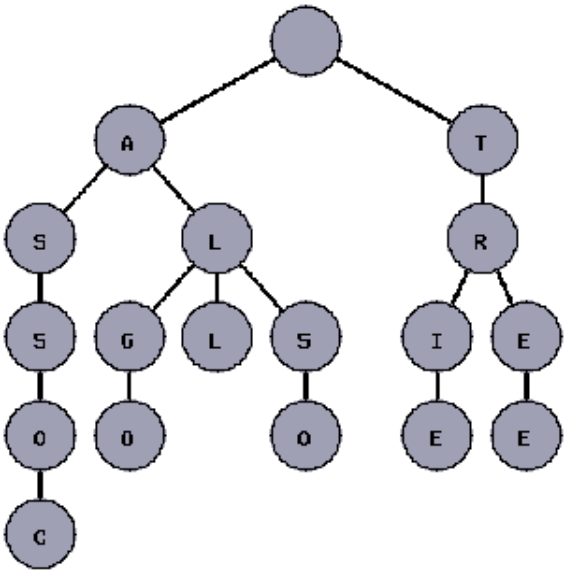

Abbildung 3.2: Beispiel eines Tries [9]

Im Folgenden soll eine formale Definition eines Tries gegeben werden:

Definition 3.1 *Sei $\sum$ eine endliche Menge von Zeichen (Alphabet) und $\sum^*$ die Menge aller Wörter, die über $\sum$ gebildet werden können. Sei $S \subseteq \sum^*$. Dann ist $T = (V, E)$ ein Trie, wobei V die Menge aller Knoten und E die Menge aller Kanten ist. Darüber hinaus muss gelten:*

- $\forall e \in E : e$ *ist mit Zeichen aus $\sum$ beschriftet.*
- $\forall v \in V$: *alle ausgehenden Kanten von v sind unterschiedlich beschriftet mit einem Zeichen $z \in \sum$*
- $\forall S_i \in S : \exists v \in V : S_i$ *ist ein Präfix der Konkatenation der Beschriftungen des Pfades vom Wurzelknoten bis v.*
- $\forall b \in V : \exists S_i \in S :$ *Die Konkatenation der Beschriftungen von der Wurzel bis b ergibt S_i, sofern b ein Blatt des Tries ist.*

Definition 2.7 ist aus [7] entnommen.

Das Suchen nach gespeicherten Wörtern gestaltet sich nun verhältnismäßig einfach: Um das Wort „Tree“ im, in der Abbildung gezeigten, Trie zu finden, wird wie folgt vorgegangen:
Stellt der User die Anfrage „Tree“, wird diese Query nun Zeichen für Zeichen durchgegangen. Beginnend bei „T“ wird im Wurzelknoten geprüft, ob es einen Verweis auf einen Knoten gibt, der ein „T“ repräsentiert [9]. Existiert ein solcher Knoten, wird in diesem geprüft, ob es einen Knoten gibt, der das nächste Zeichen in der Query (das „r“) repräsentiert. Ist dies der Fall, wird das Verfahren solange wiederholt, bis die Query komplett eingelesen ist oder in der Query ein Zeichen steht, das durch keinen Knoten im Trie repräsentiert ist [9] [5].

Ähnlich leicht funktioniert das Einfügen neuer Wörter in den Trie. Dazu wird - wie beim Suchen - das Wort, das eingefügt werden soll, so weit wie möglich nach dem oben beschriebenen Muster eingelesen und es wird zu den entsprechenden Knoten gesprungen [9]

[5]. Wird nun ein Zeichen eingelesen, das nicht durch einen Knoten repräsentiert ist, wird ein neuer Knoten erzeugt, welcher dieses Zeichen repräsentiert [9]. Alle nun noch einzulesenden Zeichen erhalten einen neuen Knoten, da der neu erzeugte Knoten natürlich nicht auf bereits vorhandene Knoten zeigen kann. Innerhalb dieses Knotens wird eine Liste angelegt, die Verweise auf weitere Knoten hält [9]. In dieser Liste wird ein Verweis auf den Knoten angelegt, der das nächste Zeichen des einzufügenden Wortes repräsentiert. Dieses Verfahren setzt sich solange fort, bis das neue Wort vollständig eingelesen ist.

Das Löschen soll in diesem Rahmen nicht aufgezeigt werden, da dies weitaus komplexer sein kann als das Finden oder Einfügen von Einträgen.

Ein Knoten, zu dem man mit dem Wort S_i gelangt, muss außerdem eine Liste mit Verweisen auf alle Dokumente, in denen das Wort S_i vorkommt, speichern. Gibt es keine Dokumente, in denen S_i vorkommt, ist die Liste leer.

4 TF-IDF

Bei großen Sammlungen von Dokumenten reicht es nicht aus, wenn das IR-System die Dokumente zurückgibt, welche die Suchanfrage erfüllen. Die Menge der Dokumente, die durch das IR zurückgegebene werden, ist meist zu groß, so dass der Nutzer nicht in der Lage ist alle Dokumente zu sichten und die für ihn relevanten Dokument auszuwählen.
Die Frage, die sich hier stellt, ist, wie der Nutzer die Dokumente bekommt, die er mit hoher Wahrscheinlichkeit benötigt. Zur Bestimmung der Wichtigkeit eines Dokuments bzgl. einer Query, wird das sogenannte Scoring eingesetzt. Scoring, oder auch Bewertung, wird genutzt um zu bestimmen, welche Dokumente für die Suchanfrage am relevantesten sind. Es kann auch von einer Gewichtung der Dokumente gesprochen werden.
Das Gewichtungsmodell, welches in dieser Arbeit thematisiert und genutzt wird, ist das TF-IDF-Maß. Das TF-IDF-Maß bewertet ein Dokument anhand der Terme, die in der Query stehen, ohne zu beachten, wie diese miteinander verknüpft sind. TF steht hierbei für Term Frequency und IDF für Inverse Document Frequency. Beide Methoden werden einzeln in den folgenden Abschnitten vorgestellt und am Ende zum TF-IDF-Maß verknüpft.

4.1 Term Frequency

Um Dokumente aus einer Kollektion nach Termen einer Suchquery zu gewichten, ist das Zählen jedes einzelnen Terms t einer Query in jedem Dokument d die zunächst trivialste Lösung. Denn ein Dokument, in dem ein Term häufiger vorkommt als in einem anderes Dokument, weist eine höhere Übereinstimmung mit diesem Term auf.

Bei dieser Vorgehensweise wird von Term Frequency (siehe Definition 4.1) gesprochen, auf deutsch auch als Suchwortdichte bekannt.

Definition 4.1 (Term Frequency) *Die Term Frequency $tf_{d,t}$ gibt die Anzahl des Terms t in dem Dokument d wieder.*

Bei dieser Vorgehensweise wird jedoch weder die Anordnung der Terme im Dokument, noch die Anordnung der Terme in der Query berücksichtigt. Die Query „Karl ist engagierter als Eckhard“ liefert die gleiche Gewichtung für die Dokumente einer Kollektion zurück wie „Eckhard ist engagierter als Karl“. Dieses Modell, das die Anordnung der Terme innerhalb des Dokumentes ignoriert, wird auch als „Bag of Words Model“ bezeichnet. Dieses interpretiert ein Dokument als eine Menge von Tupeln $\langle w_i, k \rangle$, wobei w_i das i-te Wort der Menge W ist, welche alle Wörter des Dokuments d beinhaltet. k die Häufigkeit, in der das Wort im Dokument vorkommt. Ein Dokument ist somit ein „Beutel“ mit vielen losen Wörtern. Daraus folgt, dass zwei Dokumente, welche die selben Wörter und die selbe Anzahl dieser haben, als identisch angesehen werden, obwohl sie das nicht unbedingt sind. Des Weiteren werden alle Terme als gleich wichtig gewertet, was nicht immer zielführend ist, wie zum Beispiel im Abschnitt 2.3.2 mit den „stop words“ beschrieben wurde.

4.2 Inverse Document Frequency

Da die Terme in einer Suchquery nicht alle von gleicher Bedeutung für die Suche sind, muss eine Lösung gefunden werden, die die Terme untereinander gewichtet. Wenn in einer Dokumenten-Kollektion im Automobilbereich nach „elektrische Autos sind die Zukunft“ gesucht wird, taucht das Wort Auto in allen Dokumenten häufig auf. Deshalb sollte der Term „Auto“ weniger stark ins Gewicht fallen als das Wort „elektrische“. Hier kommt die Inverse Document Frequency (siehe Definition 4.3) zum tragen. Jedoch muss vorher geklärt werden worum es sich bei der Document Frequency (siehe Definition 4.2) handelt.

Definition 4.2 (Document Frequency) *Die Document Frequency df_t gibt die Anzahl der Dokumente d in der genutzten Kollektion an, welche den Term t besitzen.*

Durch die Document Frequency kann also festgestellt werden, wie viele Dokumente einen bestimmten Term beinhalten. Die Häufigkeit, in der ein Term in den Dokumenten vorkommt, bleibt dabei unbeachtet, dafür ist die Term Frequency zuständig. Um aus der Document Frequency die einzelnen Terme sinnvoll zu gewichten, wird die Inverse Document Frequency gebildet.

Definition 4.3 (Inverse Document Frequency) *Die Inverse Document Frequency ist definiert als:*

$$idf_t = log(\frac{N_D}{df_t})$$

df_t *ist hierbei die Document Frequency und* N_D *die Anzahl der Dokumente in der Kollektion.*

Das Dividieren der Gesamtzahl aller Dokument in der Kollektion (N_D) durch die Document Frequency hat zur Folge, dass das Gewicht eines seltenen Terms höher ist und das eines häufigen Terms niedrig. Der Logarithmus sorgt dafür, dass das resultierende Gewicht nicht zu groß wird, wenn die Kollektion sehr groß und der Term selten ist.

4.3 Das TF-IDF-Maß

In diesen Abschnitt wird die Zusammensetzung der Term Frequency und der Inverse Document Frequency zum TF-IDF-Maß beschrieben.

Definition 4.4 (TF-IDF) *Das TF-IDF-Maß ist wie folgt definiert:*

$$tf\text{-}idf_{t,d} = tf_{t,d} * idf_t$$

Hierbei wird lediglich das Produkt der Inverse Document Frequency mit der Term Frequency gebildet.

Nun kann eine score-Funktion definiert werden, die einem Dokument ein Gewicht zuweist. Die folgende Funktion $score(q, d)$ zeigt die Lösung, wobei q hier die Suchquery ist und d das zu gewichtende Dokument:

$$score(q, d) = \sum_{t \in q} tf\text{-}idf_{t,d}$$

Wie in der Funktion beschrieben, wird für jeden Term in der Suchquery das TF-IDF-Maß berechnet. Die einzelnen Gewichte werden dann aufsummiert. Am Ende dieser Prozedur erhält man dann eine Gewichtung für ein Dokument anhand der Suchquery und ist in der Lage seine Relevanz mit der Relevanz der anderen gefundenen Dokumente zu vergleichen.

5 Implementierung - Lokale Suchmaschine

5.1 Ziel der Beispiel-Implementierung

Im Folgenden wird eine Beispiel-Implementierung der zuvor theoretisch diskutierten Inhalte vorgestellt. Dabei wird eine lokale Suchmaschine entwickelt, welche in der Lage ist, PDF-Dateien auf einem lokalen Computer-System zu parsen, in einen invertierten Index aufzunehmen, sowie Suchanfragen eines Benutzers sinnvoll zu beantworten. Zur Relevanz-Bestimmung der Dokumente wird das TF-IDF-Maß, welches bereits vorgestellt wurde, genutzt. Das Speichern des Index wird mit der von Python mitgelieferte Datenstruktur *Dictionary*, welche im Grunde eine Hashmap ist, umgesetzt. Weiter werden Bibliotheken eingesetzt, welche einige Vorarbeit leisten und damit den Code der Beispiel-Implementierung auf das Wesentliche beschränken. Die Anwendung soll die grundlegende Arbeitsweise eines Information Retrieval-Systems darlegen.

5.2 Genutzte Bibliotheken

Vor der eigentlichen Implementierung der lokalen Suchmaschine werden einige Module eingebunden, welche die Implementierung unterstützen. Folgend werden die Module aufgelistet und wichtige Module im jeweiligen Abschnitt genauer erläutert:

- Apache Tika: Erkennt und extrahiert Metadaten und Texte aus über tausend Dateitypen
- Math: Standard Python-Modul für mathematische Funktionen
- OS: Stellt Betriebssystem-Funktionalitäten bereit. Wird hier genutzt um durch Verzeichnisse zu navigieren
- filetype: Wird zur Dateityp-Erkennung genutzt
- re: Das re-Modul stellt Funktionen bereit, mit denen mit regulären Ausdrücken gearbeitet werden kann. In der Beispiel-Implementierung werden mit diesem Modul die Tokens ermittelt
- Platform: Prüfung, auf welchem Betriebssystem das Programm läuft, da Windows- und Linux-Systeme verschiedene Dateisysteme nutzen
- Operator: Dieses Modul exportiert effiziente Funktionen, die den eigentlichen Operatoren von Python entsprechen. Es findet nur Anwendung in der Sortierung der zu zurückgebenden Dokumente in der retrieve-Methode.
- NLTK: Stellt Funktionen zur Verfügung, die es ermöglichen mit menschlichen Sprachdaten zu arbeiten.

5.2.1 Apache Tika

Bei Apache Tika handelt es sich um eine Bibliothek, um Inhalte aus Dateien zu erkennen und zu analysieren. Es ist in der Lage Text und Metadaten aus verschiedenen Arten von Dateien zu extrahieren. Apache Tika liefert einen Parser, mit dessen Hilfe der Text aus - unter anderem - PDF-Dateien extrahiert werden kann. Mit dem Aufruf *parser.from_file(file)* kann ein PDF-Dokument in Text umgewandelt werden.

5.2.2 filetype

Mittels filetype ist es möglich, unabhängig von der Dateiendung, den Typ einer Datei zu ermitteln. Dies hat den Vorteil, dass die Suchmaschine sowohl unter Windows, als auch

```
from tika import parser
import filetype
import math
import os
import string
import platform
import operator
import re
from nltk.tokenize import RegexpTokenizer
```

Abbildung 5.1: Imports

unter Unix-Systemen, alle PDF-Dateien finden kann, da unter Unix die Dateiendung keine garantierten Rückschlüsse auf den Typ der Datei zulässt.

5.2.3 NLTK

NLTK (natural language toolkit) ist eine Bibliothek für Python, die für die Verarbeitung natürlicher Sprachen eingesetzt wird. Dabei bietet die Bibliotheken Funktionen für unter anderem Textklassifikation, Tokenization und Stemming. In diesem Beispiel wird nltk verwendet, um die Eingabetexte der Dokumente und die Eingaben des Nutzers zu Tokens zu verarbeiten. Dazu wird eine Klasse verwendet, die auf Basis von Regular Expressions arbeitet. Mehr dazu wird anhand der Implementierung gezeigt.

5.2.4 Platform

Das Platform-Modul wird in dieser Implementierung dazu verwendet, festzustellen, auf welchem Betriebssystem die lokale Suchmaschine ausgeführt wird. Dies muss ermittelt werden, da auf Windows und Linux unterschiedliche Dateisysteme arbeiten. In Linux ist das Startverzeichnis immer das Root-Verzeichnis („/“). Unter Windows gibt meist mehrere Partitionen, die alle durchsucht werden müssen.

```
class Document:

  def __init__(self, url, length, docId, termList):
    self.url = url
    self.length = length
    self.docId = docId
    self.score = 0.
    self.termList = termList
```

Abbildung 5.2: Dokumentenklasse

5.3 Die Document-Klasse

Das Speichern der für das Retrieval wichtigen Informationen, geschieht mittels einer Document-Klasse. Diese Klasse hält alle Member-Variablen, die wichtig sind, um das TF-IDF-Maß berechnen zu können. Diese Member-Variablen sind:

- url: String mit dem Pfad zum Dokument, welches von dieser Instanz repräsentiert wird
- length: Integer, welcher die Anzahl der Wörter in diesem Dokument darstellt
- docId: Integer, welcher dem Dokument einen eindeutigen Identifikator gibt
- score: Float, welcher die Gewichtung der Dokument-Instanz zu einer Anfrage angibt
- termList: Eine Liste von Strings, die die Terme des Dokumentes entsprechen. Diese wird durch die Methode *_preprocess* erzeugt (siehe Abschnitt 5.4.2)

Die Member-Variable *url* soll am Ende des Retrievalprozesses zurückgegeben werden, da der Nutzer durch den Pfad direkten Zugriff auf das Dokument bekommt. Die Member-Variablen *length*, *termList* und *score* werden für die Berechnung des TF-IDF-Maßes benötigt und so auch für den Retrievalprozess. Die Member-Variable *docId* wird in der Indexklasse einerseits genutzt, um die Identifier in dem *invIndex*-Dictionary den jeweiligen Termen zuzuordnen. Des Weiteren erfolgt die Zuordnung der jeweiligen ID zu ihrer Document-Instanz über die *docHashmap* in der Indexklasse (siehe invIndex und docHashmap in Abschnitt 5.4).

```
def tf_idf(self, terms, df, fileCount):

  tfDict = {}
  for term in terms:
    tfDict[term] = 0

  for term in self.termList:
    if term in terms:
      tfDict[term] = tfDict[term]+1

  for key, value in df.items():
    idf = math.log((fileCount/value+1),10)
    tfDict[key] = tfDict[key]*idf

  self.score = sum(tfDict.values())

Document.tf_idf = tf_idf
```

Abbildung 5.3: TF-IDF-Methode

5.3.1 TF-IDF

Die Document-Klasse hält neben den benötigten Attributen auch die Scoringmethode *tf_idf*. Dies ist die Implementierung des TF-IDF-Maßes und berechnet für jedes Dokument die Gewichtung für eine gegebene Suchquery aus. Hierbei benötigt die Funktion die Terme der Suchquery, welche über das Attribut *terms* übergeben werden. Das Attribut *df* steht für die Document Frequency, welches für jeden Term die Anzahl der gefunden Dokumente in Form eines Dictionaries beinhaltet. Über das Attribut *fileCount* wird die Anzahl der Dokumente in der Kollektion, hier die Dateien die im invertierten Index aufgenommen wurden, übergeben.

Als erstes wird das Dictionary *tfDict* für die Term Frequency erstellt und mit den Termen der Query als Schlüssel und den Werten 0 initialisiert. In der nächsten For-Schleife wird über *self.termList* iteriert, welches die Liste der Terme des Dokumentes sind. Wenn ein Term auch in der Suchquery enthalten ist, also *term in terms*, dann wird im *tfDict* der Wert des gefunden Terms um eins aufaddiert. Am Ende enthält das *tfDict* die Terme der Suchquery als Schlüssel und die Anzahl ihrer Vorkommnisse im Dokument als Wert, also die Term Frequency. Als letztes muss jede Term Frequency mit der Inverse Document Frequency multipliziert werden. Dafür wird über das Dictionary *df* iteriert und für jedes

```
class Index:

  def __init__(self):
    self.invIndex = {}
    self.fileCount = 0
    self.docHashmap = {}
```

Abbildung 5.4: Indexklasse

Key-Value-Paar die Inverse Document Frequency ausgerechnet und auf die jeweilige Term Frequency multipliziert. Am Ende wird die Summe aller TF-IDF-Maße der *score*-Variable zugeordnet und bilden so die finale Gewichtung für die gegebene Suche.

5.4 Der Index

Die Index-Klasse beinhaltet alle Methoden um den invertierten Index aufzubauen und die *retrieve*-Methode, welche die Dokumente, sortiert nach deren Gewicht, zu einer gegebenen Query zurückgibt. Folgenden Member-Variablen werden für den invertierten Indexaufbau und die *retrieve*-Methode benötigt:

- invIndex: Ist ein Dictionary mit einem Term als Schlüssel und einer Menge von Document IDs als Wert
- fileCount: Zählt beim Aufbau des invertierten Indexes die Dokumente, die in diesem aufgenommen werden
- docHashmap: Ist ein Dictionary welches eine Document ID ihrer zugehörige Document-Klasseninstanz zuordnet

Die Member-Variable *invIndex* ist der invertierte Index welcher bei der *retrieve*-Methode die Document IDs zu einen gegebenen Term aus der Suchquery zurückgibt. Des Weiteren wird die Member-Variable *docHashmap* dafür genutzt, um über die Document IDs auf die jeweiligen Dokumentinstanzen zuzugreifen und so auf ihren Inhalt und die Scoringfunktion *tf_idf*. Die Member-Variable *fileCount* gibt die Größe der Kollektion wieder und wird bei der Berechnung des TF-IDF-Maßes benötigt.

5.4.1 buildIndex

Diese Methode *buildIndex* baut den invertierten Index auf und nutzt dafür die *_getStartDirectories*- und die *_addToIndex*-Methode. Dabei kann der *buildIndex*-Methode optional Startverzeichnisse in Form einer Liste mit raw Strings mitgegeben werden. In der ersten If-Abfrage wird geprüft, ob die Liste *startDirectories* leer ist. Dies ist der Fall, wenn der Nutzer keine Startverzeichnisse mitgegeben hat. Dann werden die Startverzeichnisse mithilfe der *_getStartDirectories*-Methode ermittelt.

Der erste Schritt stellt das Iterieren über alle Startverzeichnisse der Liste *startDirectories* dar. Anschließend werden für das Startverzeichnis, und alle darunterliegenden Verzeichnisse, bis zur untersten Ebene, die Dateien mithilfe der *os.walk*-Funktion geholt. Für jede Datei wird dann mit den Funktionen *os.path.abspath* und *os.path.join* der absolute Pfad gebildet. Danach wird durch das Modul *filetype* ermittelt, ob es sich um ein PDF-Dokument handelt. Ist ein Dokument vom Typ PDF, wird mithilfe des *Tika*-Moduls, genauer gesagt mit dem *parser* der Text aus dem PDF-Dokument extrahiert. Dies geschieht in dem der Funktion *from_file* des Parsers mit dem Pfad der Datei aufgerufen wird. Bei der Rückgabe kann mithilfe des Schlüssels *content* auf den Text zugegriffen werden, welcher dann in der Variable *rawText* gespeichert wird.

Für jede entdeckte PDF-Datei wird die Member-Variable *fileCount* um eins erhöht. Da die Zahl sich bei jedem gefunden Dokument ändert, wird diese gleich als Document ID genutzt, da sie für jedes Dokument eindeutig ist. Im Folgenden Schritt wird der *rawText* mithilfe der *_preprocessText*-Methode normalisiert und in eine Liste von Tokens geteilt und in der Liste *processedText* gespeichert. Nun sind alle Daten vorhanden um eine Documentinstanz zu erstellen. Diese enthält, wie in Abschnitt 5.3 schon beschrieben, den Pfad zum Dokument, die Anzahl der Wörter, die Document ID und den Text als Liste von Tokens.

Im vorletztem Schritt wird die erstellte Documentinstanz seiner Document ID in der Member-Variable *docHashmap* zugeordnet, um später auf diese Instanz zurückgreifen zu können. Als letztes wird der invertierte Index, mithilfe der *_addToIndex*-Methode, der Tokenliste *processedText* und der Document ID, aktualisiert.

```
def buildIndex(self, startDirectories=[]):

  if not startDirectories:
    startDirectories = self._getStartDirectories()

  for directory in startDirectories:
    for root, _, files in os.walk(directory):
      for file in files:

        path = os.path.abspath(os.path.join(root, file))

        try:
          if filetype.guess(path).mime == "application/pdf":

            fileData = parser.from_file(path)
            rawText = fileData['content']
            self.fileCount += 1

            processedText = self._preprocessText(rawText)
            document = Document(path, len(processedText),
                        self.fileCount, processedText)
            self.docHashmap.update({self.fileCount : document})
            self._addToIndex(self.fileCount, processedText)
        except:
          print(end="")
          continue

Index.buildIndex = buildIndex
```

Abbildung 5.5: BuildIndex-Methode

```
def _getStartDirectories(self):

  if platform.system() == "Linux":
    directories = ["/"]

  elif platform.system() == "Darwin":
    directories = ["/"]

  elif platform.system() == "Windows":
    directories = ['%s:\\' % d for d in string.ascii_uppercase
                   if os.path.exists('%s:' % d)]

  else:
    raise EnvironmentError

  return directories

Index._getStartDirectories = _getStartDirectories
```

Abbildung 5.6: GetStartDirectories-Methode

5.4.2 Hilfsmethoden

In diesem Abschnitt werden die drei Hilfsmethoden *_getStartDirectories*, *_preprocessText* und *_addToIndex* vorgestellt, welche in der *buildIndex*-Methode genutzt werden.

_getStartDirectories

Die Methode *_getSartDirectories* liefert eine Liste der Start-Verzeichnisse, abhängig vom Betriebssystem auf dem die Suchmaschine läuft. In diesen Verzeichnissen werden dann in der *buildIndex*-Methode rekursiv nach PDF-Dateien gesucht, welche in den Index mit einfließen. Falls das zugrunde liegende Betriebssystem ein Linux-basiertes oder Mac-basiertes System ist, wird die Liste [„/“] zurückgegeben, da das Verzeichnis / immer das Root-Verzeichnis ist. Bei einem auf Windows basierenden Systemen gibt es wiederum mehrere Partitionen, welche immer mit einem Großbuchstaben abgekürzt werden. Dementsprechend gibt es unter Windows auch mehrere Root-Verzeichnisse.

Zuerst wird mithilfe der *system*-Funktion des *platform*-Moduls geprüft welches Betriebssystem vorliegt. Bei einem auf Linux- (Rückgabe „Linux“)oder Mac-basierenden System (Rückgabe „Darwin“) wird einfach eine List mit dem Element „/“ erstellt.

```
def _addToIndex(self, documentID, terms):

  for term in terms:

    try:
      docSet = self.invIndex[term]
      docSet.add(documentID)
      self.invIndex.update({term : docSet})

    except KeyError:
      docSet = {documentID}
      self.invIndex.update({term : docSet})

Index._addToIndex = _addToIndex
```

Abbildung 5.7: AddToIndex-Methode

Bei einem auf Windows basierenden Betriebssystem (Rückgabe „Windows“) ist das Erstellen der Startverzeichnisse aufwändiger. Hierbei werden alle Großbuchstaben, im Code durch *string.ascii_uppercase* aufrufbar, darauf geprüft eine Partition zu sein. Dies wird mithilfe der *os.path.exists*-Methode realisiert. Ist ein Großbuchstabe tatsächliche eine Partition auf dem Computer, so wird er in der Liste gespeichert. Jedoch wird an den Großbuchstaben noch der String „:\“ angehangen, damit die *buildIndex*-Methode mit den Elementen der Liste als Start-Verzeichnisse arbeiten kann.

_addToIndex

Die Methode *_addToIndex* soll die Dokumenten ID zu den invertierten Index der in dem Dokument vorkommenden Terme hinzufügen. Hierfür bekommt die Methode eine Liste von Termen die in einem PDF-Dokument vorkommen über den Parameter *terms* und die zum Dokumente gehörige Document ID als Parameter *documentID* übergeben.

Für jeden Term in der Liste *terms* wird dazu der zum Term gehörige Eintrag im invertierten Index nachgeschlagen. Schlägt der Versuch fehl, da noch kein Eintrag des Terms in dem invertierten Index vorhanden ist, wird ein neuer Eintrag für diesen Term und der übergebenen Document ID erstellt. Wird ein Eintrag gefunden, wird die übergebende Document ID der schon vorhandene Menge hinzugefügt und der invertierte Index wird geupdatet.

_preprocessText

Diese Methode dient der Vorverarbeitung der Texte, die in den PDF-Dokumenten stehen, wird aber auch für die Suchquery genutzt. Hierfür wird der Text eines PDF-Dokumentes an den Parameter *text* übergeben. Als erster Schritt wird der gesamte Text in Lower-Case (Kleinschreibung) gesetzt, damit später bei der Suche die Groß- bzw. Kleinschreibung irrelevant ist. Das Ergebnis wird in der Variable *lowerText* gespeichert. Im nächsten Schritt werden alle Zahlen aus *lowerText* entfernt und in *prepText* gespeichert, da Zahlen für die Textsuche nicht von Bedeutung sind.

Als nächstes wird mithilfe der Klasse *RegexpTokenizer*, die durch die NLTK-Bibliothek zur Verfügung gestellt wird, der String *prepText* in eine Liste von Tokens aufgespalten. Was als Token gewertet wird, wird mithilfe einer *regular expression* definiert, im Deutschen regulärer Ausdruck genannt. Ein regulärer Ausdruck ist eine Zeichenkette, welche eine Menge von bestimmten Zeichenketten beschreibt. Der gewünschte reguläre Ausdruck wird dem Konstruktor der *RegexpTokenizer*-Klasse in Form eines raw Strings übergeben. Ein raw String ist ein String, welcher mit einem *r* am Anfang gekennzeichnet ist und ein Backslash als ein Literal und kein Escape-Zeichen behandelt. Dies ist bei regulären Ausdrücken nützlich, da in diesen viel mit Backslashs gearbeitet wird.

Im Folgenden wird der raw String bzw. reguläre Ausdruck näher betrachtet, um zu verstehen, was als ein Token gewertet wird. Der erste Teil des regulären Ausdrucks *[a-zA-Z]+-$* definiert alle Buchstabenketten mit einen oder mehreren Elementen, die mit einem Bindestrich bei einem Zeilenumbruch enden, als Token. Die eckigen Klammern werden bei regulären Ausdrücken genutzt, um eine Zeichenauswahl zu definieren. Das bedeutet, dass ein Zeichen aus dieser Auswahl dann an dieser Stelle steht. Mithilfe von Quantoren kann definiert werden, wie viele Zeichen einer Auswahl hintereinander stehen dürfen. Das Pluszeichen ist genau so ein Quantor, welcher aussagt, dass mindestens ein oder mehrere Zeichen der Zeichenauswahl hintereinander vorkommen muss. Die Funktion des Dollarzeichens hängt von einem Flag, dem Multiline-Flag ab. Dieses ist in NLTK standardmäßig gesetzt und bewirkt, dass das Dollarzeichen für das Ende einer Zeile steht.[8] Sonst würde das Dollarzeichen das Ende eines Wortes markieren. Dadurch das ein Bindestrich vor das Dollarzeichen des regulären Ausdrucks gesetzt haben, bedeutet der reguläre Teilausdruck

-$, dass die letzte Zeichenkette einer Zeile auf einem Bindestrich endet.

Bei dem |-Zeichen handelt es sich um eine logische Oder-Verknüpfung, die es ermöglicht, mehrere reguläre Ausdrücke zu verknüpfen. In unserem Fall *\w+*. Der zweite reguläre Ausdruck *\w+* definiert alle alphanumerischen Zeichenketten mit einem oder mehreren Elementen als Token. Hierbei ist *\w* eine vordefinierte Zeichenklasse für den regulären Ausdruck *[a-zA-Z_0-9]* und beinhaltet außer alphanumerische Werte auch noch den Unterstrich. Das Pluszeichen ist hier wieder der Quantor, welcher aussagt, dass aus dieser Zeichenklasse ein oder mehrere Zeichen hintereinander vorkommen muss. Mithilfe der *tokenize*-Methode wird der reguläre Ausdruck auf den String *prepText* angewendet. Jeder Substring des mitgegebenen Strings, der den regulären Ausdruck erfüllt, wird an die Liste *tokenList* angefügt.

Der Grund warum die Wörter, die auf einem Bindestrich enden, bei der Tokenerzeugen extra beachtet werden, ist der, dass die Wörter, welche bei Zeilenumbrüchen getrennt werden, wieder zusammengefügt werden sollen. In der for-Schleife werden diese Tokens auf die Eigenschaft hin, auf einem Bindestrich zu enden, geprüft und gegebenenfalls zusammengesetzt. Dazu wird der Bindestrich aus dem Token entfernt und mit dem nächsten Token in der Tokenliste verknüpft (*token[:-1]+tokenList[index+1]*). Die Tokenliste wird darauf hin aktualisiert. Der neu zusammengesetzte Token ersetzt den Token, welcher den Bindestrich enthielt, und der darauffolgende Token der Liste wird gelöscht, da er schon mit dem Token davor zusammengesetzt wurde.

Diese Methode, die Wörter mit einem Bindestrich am Zeilenende mit dem nächsten Token zusammenzufügen, ist jedoch nicht immer korrekt. Es kann auch folgender Fall eintreten: Eine Zeile endet zum Beispiel mit *Damen-* und die nächste Zeile geht mit *und Herrenschuhe* weiter. In diesem Fall ist der Bindestrich gewollt, der Algorithmus fügt jedoch die Wörter *Damen* und *und* zu einem Wort zusammen. Da davon auszugehen ist, dass dieser Fall selten eintritt, wurde er vernachlässigt.

```
def _preprocessText(self, text):

  lowerText = text.lower()

  prepText = re.sub(r'\d+', '', lowerText)

  tokenizer = RegexpTokenizer(r'[a-zA-Z]+-$|\w+')
  tokenList = tokenizer.tokenize(prepText)

  for token in tokenList:
    if token[-1] == '-':

      index = tokenList.index(token)
      compositeWord = token[:-1]+tokenList[index+1]
      tokenList[index] = compositeWord
      del tokenList[index+1]

  return tokenList

Index._preprocessText = _preprocessText
```

Abbildung 5.8: PreprocessText-Methode

5.4.3 retrieve

Die *retrieve*-Methode dient der Suche nach Dokumenten anhand einer eingegebenen Suchquery und stellt die Schnittstelle zum Nutzer dar. Die Idee dabei ist, dass der Nutzer ein oder mehrere Schlagworte, in Form eines Strings, der Methode übergeben kann. Auf deren Basis werden die gefundenen Dokumente, nach Relevanz sortiert, zurückgeliefert.

Zunächst wird der Suchstring des Nutzers mittels der bereits bekannten Methode *_preprocessText* normalisiert und in eine Liste von Termen zerteilt. Das Ergebnis wird in der Variable *processedStrings* abgespeichert. Im zweiten Schritt werden alle genutzten Variablen deklariert. Mithilfe der For-Schleife wird über die Liste von Termen *processedStrings* iteriert, um über den invertierten Index für jeden Term die Menge der Document IDs zu beschaffen. Desweiteren wird für jeden Term die Länge seiner Menge von Document IDs im Dictionary *df* abgespeichert und repräsentiert so die Document Frequency. Die Variable *result* stellt die Menge da, die alle zur Suchquery gefundenen Document IDs enthält. So wird diese Variable mit jeder Menge von Document IDs vereinigt. Da es sich hier um eine Menge handelt kann ausgeschlossen werden, dass Document IDs doppelt vorkommen. Existiert der Term *word* nicht als Key im Index, da der Nutzer ein Wort eingegeben hat,

```
def retrieve(self, searchString):

  processedStrings = self._preprocessText(searchString)
  result = set()
  df, weightedDocs = {}, {}
  resultList = []

  for word in processedStrings:
    try:
      documents = self.invIndex[word]
      df[word] = len(documents)
      result = result.union(documents)
    except KeyError:
      continue

  for document in result:
    doc = ind.docHashmap[document]
    doc.tf_idf(processedStrings,df,self.fileCount)
    weightedDocs[doc.docId] = doc.score

  sortedDocs = sorted(weightedDocs.items(), key=operator.itemgetter(1))

  for key,_ in sortedDocs:
    resultList.append(ind.docHashmap[key].url)

  return resultList[::-1]

Index.retrieve = retrieve
```

Abbildung 5.9: Retrieve-Methode

welches kein Dokument beinhaltet, wird beim nächsten Term der Liste *processedStrings* fortgefahren.

In der nächsten For-Schleife wird für jede Document ID in der Liste *result*, mithilfe der *docHashmap*, die passende Documentinstanz geholt, um für jedes Dokument das TF-IDF-Maß auszurechnen. Wurde dies mithilfe der *tf_idf*-Methode getan, wird das Gewicht zu der jeweiligen Document ID in dem Dictionary *weightedDocs* gespeichert. Dieses wird mit der *sorted*-Funktion von Python nach den Gewichten sortiert. Die Sortierung ist jedoch aufsteigend, obwohl es gewünscht ist, die Dokumente mit einem höheren Gewicht oben zu haben. Dafür wird die Liste am Ende einfach invertiert.

In der letzten For-Schleife werden die absoluten Pfade (*ind.docHashmap[Key].url*), in der sortierten Reihenfolge, in die Liste *resultList* gespeichert. Diese wird dann invertiert, wegen des oben genannten Grundes, und zurückgegeben.

```
ind = Index()
ind.buildIndex([r"Opt\Start\1",r"Opt\Start\2",...])

resultSet = ind.retrieve("This␣is␣a␣test")
if resultSet:
  for elem in resultSet:
    print(elem)
```

Abbildung 5.10: Execute

5.5 Ausführung der Suchmaschine

In diesem Abschnitt soll kurz erläutert werden, wie die oben erstellten Klassen genutzt werden. Als erstes muss eine neue Instanz der Indexklasse erstellt werden. Über diese Instanz wird dann die *buildIndex*-Methode aufgerufen. Optional kann hier eine Liste der gewünschten Startverzeichnisse als raw Strings übergeben werden. Wenn der invertierte Index aufgebaut wurde, kann über die Indexinstanz die *retrieve*-Methode mit der gewünschten Suchquery aufgerufen werden. Diese liefert eine nach Gewichten sortierte Liste, mit den Pfaden zu den gefundenen Dokumenten. Wenn diese nicht leer ist, können mithilfe einer For-Schleife die Elemente ausgegeben werden.

6 Verbesserungen

In der Vorgestellten Implementierung gibt es einige Probleme und daraus folgende Verbesserungen, die in diesem Abschnitt besprochen werden.

Die folgende Liste dient nur zur Dokumentation der vorhanden Probleme und wird später durch ausführlichere Erläuterungen und Beispielen ersetzt.

- BuildIndex Laufzeit verringern
  - Mit Threads parallelisieren?
- Invertierten Index persistieren, damit er nicht immer wieder neu aufgebaut werden muss
  - Wie soll persistiert werden?
  - Wie sollen nur Verzeichnisse mit Änderungen (Löschung/Erstellung von PDF) erkannt werden?
  - Wann sollen Aktualisierungen durchgeführt werden?
- *.dat Dateien werden als PDFs erkannt
- Implementierung von Stemming?
- RAM-Auslastung durch Speicherung der Terme der Dokumente als Liste in Document-Klasse hoch. Verlangsamt BuildINdex, beschleunigt aber Retrieve.
- Kontext der Wörter wird nicht beachtet (Bags of Words Modell)
  - N-Gramm nutzen? (Speichern von zwei aufeinander treffende Wörter als Token)

6.1 Probleme der aktuellen Beispiel-Implementierung

Dieser Abschnitt wird eine kurze Problemanalyse der aktuellen Implementierung der lokalen Suchmaschine liefern. Zu diesem Zweck soll die NPL-Test-Collection herangezogen werden. Mit deren Hilfe kann ermittelt werden, wie „gut“ die Suchmaschine ist. Dazu werden zwei Messungen vorgenommen:

- Die $precision$ gibt den Anteil an, wie viele der gefundenen Dokumente relevant sind.
- Der $recall$ gibt an, wie viele der relevanten Dokumente gefunden werden.

6.1.1 Analyse der Precision

Die Precision ist wie folgt definiert:

$$precision = \frac{r_1}{r_2},$$

wobei r_1 die Anzahl der ermittelten, relevanten Dokumente ist und r_2 die Anzahl aller ermittelten Dokumente.
In der NPL-Collection sind 11.429 Dokumente vorhanden, zudem sind 93 Queries mitgeliefert. Den Queries liegen darüber hinaus alle Dokumenten-IDs bei, die für die Query relevant sind. Falls die Suchmaschine genau die einer Query zugeordneten Dokumenten zurückliefert, wird die $precision$ 1. Wird der Wert der $precision \geq 0,8$, ist das Ergebnis bereits ausreichend gut.

Test und Auswertung der Ergebnisse

Lösungsvorschläge

6.1.2 Analyse des Recalls

Der Recall ist definiert als

$$recall = \frac{r_1}{r_2},$$

wobei r_1 die Anzahl der ermittelten, relevanten Dokumente ist und r_2 die Anzahl relevanter Dokumente ist. Der $recall$ wird 1, wenn die Menge der zurückgegebenen Dokumente genau der Menge der relevanten Dokumenten in der Collection entspricht. Gilt $recall \geq 0,8$, ist der Wert ausreichend gut.

Test und Auswertung der Ergebnisse

Lösungsvorschläge

6.1.3 Aufbaudauer des Index

Der letzte Punkt, der ins Auge fällt, betrifft die Geschwindigkeit, in der der Index aufgebaut wird. Für 189 pdf-Dokumente benötigt die Beispiel-Implementierung ca 1 Minute und 17 Sekunden. Wird ein System mit ca. 105GB nach pdf-Dateien durchsucht und in den Index aufgenommen, werden auf einem System mit einem Intel Core i5 der 8. Generation und 8 Gigabyte RAM ca. 40 Minuten benötigt. Für einen sinnvollen Einsatz einer lokalen Suchmaschine ist die Suchmaschine somit nicht geeignet. Zwei Ideen, diese Zeit zu senken, sind Parallelisierung und Persistierung.

Persistierung

Die Idee, die wahrscheinlich am wirksamsten ist, besteht darin, den Index nach oder bereits während des Aufbaus, zu persistieren. Durch die Persistierung kann bei einem Neustart der Suchmaschine eine Datei eingelesen werden, welche die Index-Daten beinhaltet. Dadurch ist es nicht nötig, beim Start durch alle Pfade zu gehen und jede Datei zu öffnen, um festzustellen, ob es sich um eine pdf-Datei handelt. Pro Öffnen einer Datei werden mindestens zwei Systemaufrufe benötigt (öffnen und schließen der Datei). Zudem werden weitere Systemaufrufe benötigt, wenn Verzeichnisse geöffnet werden, da diese in der Regel ebenfalls als spezielle Dateien behandelt werden. Das hat zur Folge, dass viele Kontextwechsel stattfinden und das Programm bei jedem Systemaufruf blockiert wird.
Dadurch, dass bei einem persistenten Index initial nur eine Datei geöffnet werden muss, kann ein Großteil der Systemaufrufe und Kontextwechsel verhindert werden. Damit wird

viel Zeit gespart.
Ein Problem, was hier gelöst werden muss ist, alle Dokumente ausfindig zu machen, die noch nicht im Index aufgenommen wurden. Dazu kann eine Liste angelegt werden, die alle im Index vorhandenen Dokumente aufführt. Zudem kann für dieses Problem Parallelisierung genutzt werden, die im Folgenden Abschnitt behandelt wird.

Parallelisierung

Durch Nutzung von Parallelisierung kann der Aufbau des Index ebenfalls beschleunigt werden. Da die Beispiel-Implementierung auf Python basiert, ist jedoch zu beachten, dass Python-Threads keine Vorteile Multi-Prozessorsystemen ziehen kann. Das bedeutet, dass mehrere Python-Threads nicht parallel auf verschiedenen Kernen laufen können. Dennoch bringt Parallelisierung einen Geschwindigkeits-Vorteil.
Wird ein Systemaufruf ausgelöst, um beispielsweise eine Datei zu öffnen, wird, sofern die Suchmaschine Single-Threaded ist, das komplette Programm blockiert. Werden jedoch Threads genutzt, kann nur der Thread blockiert werden, der den Systemaufruf verursacht hat. An dessen Stelle kann dann ein weiterer Thread treten, der in der Zwischenzeit rechnen kann. Dadurch wird weniger Zeit mit Warten verschwendet und der Index-Aufbau wird beschleunigt.
Ist der Index persistent gespeichert, ist es möglich, in regelmäßigen einen oder mehrere Threads zu starten und somit „im Hintergrund“ nach neuen Dateien gesucht werden, die noch nicht im Index vorhanden sind.
Durch die Kombination beider Ideen, sollte der Aufbau des Index spürbar beschleunigt werden.

7 Fazit

Literatur

[1] Nagy Istvan. *Indexierung mittels invertierter Dateien*. 2001. URL: http://www.cis.uni-muenchen.de/people/Schulz/SeminarSoSe2001IR/Nagy/node4.html.

[2] Ingo Frommholz. *Information Retrieval*. 2007. URL: http://www.is.informatik.uni-duisburg.de/courses/ie_ss07/folien/folien-ir.pdf.

[3] Andreas Henrich. *Information Retrieval 1*. 2008.

[4] Hinrich Schütze Cristopher D. Manning Prabhakar Raghavan. *Introduction to Information Retrieval*. 2009.

[5] Vaidehi Joshi. *Trying to Understand Tries*. 2017. URL: https://medium.com/basecs/trying-to-understand-tries-3ec6bede0014.

[6] *Vektorraum-Retrieval*. 2017. URL: https://de.wikipedia.org/wiki/Vektorraum-Retrieval.

[7] o.A. *Trie*. 2018. URL: https://de.wikipedia.org/wiki/Trie.

[8] o.V. *ource code for nltk.tokenize.regexp*. 17. Nov. 2018. URL: https://www.nltk.org/_modules/nltk/tokenize/regexp.html.

[9] freeCodeCamp. *Introduction to Trie*. URL: https://guide.freecodecamp.org/miscellaneous/data-structure-trie/.

[10] Karin Haendelt. *Klassische Information Retrieval Modelle Einführung*.