Information Retrieval

Seminararbeit

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

von

Jesse-Jermaine Richter, Jonas Seng

DD.MM.JJJJ

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 [9]. 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.

2.4 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.

2.4.1 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 2.1: Beispiel für einen invertierten Index [1]

2.4.2 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.

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.

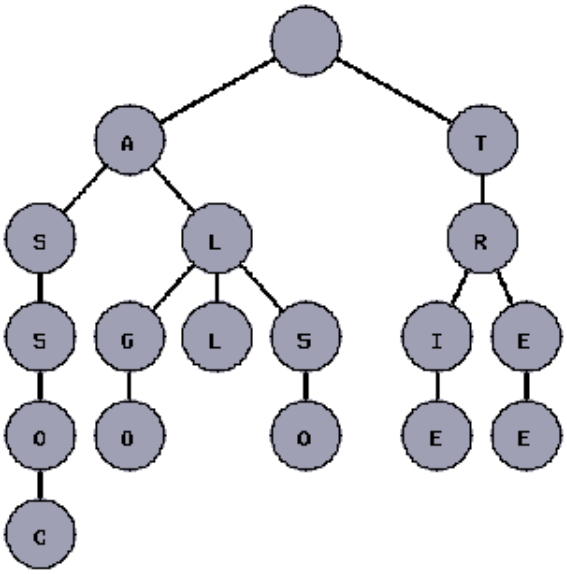

Abbildung 2.2: Beispiel eines Tries [8]

Im Folgenden soll eine formale Definition eines Tries gegeben werden:

Definition 2.7 *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 [8]. 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 [8] [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 [8]

[5]. Wird nun ein Zeichen eingelesen, das nicht durch einen Knoten repräsentiert ist, wird ein neuer Knoten erzeugt, welcher dieses Zeichen repräsentiert [8]. 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 [8]. 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.

2.5 TF-IDF Gewichtung

2.6 Retrieval

3 TF-IDF

Bei großen Sammlungen von Dokumenten reicht es nicht aus, wenn das IR die Dokumente, welche die Suchanfrage erfüllen, zurück gibt. Die Menge der Dokumente, die durch das IR zurückgegebene werden, ist meist so groß, 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 wahrscheinlichsten benötigt. Hier fällt der Begriff des Scorings. Scoring bedeutet so viel wie Bewertung und wird genutzt um zu bestimmen, welche Dokumente für die Suchanfrage am relevantesten ist. Es kann auch von einer Gewichtung der Dokumente gesprochen werden.
Das Gewichtungsmodell, welches in dieser Arbeit thematisiert und genutzt wird ist das TF-IDF-Modell. TF steht hierbei für Term Frequency und bedeutet so viel wie oft ein Term in

4 Implementierung

```
from IPython.core.display import HTML, display
display(HTML('<style>.container␣{␣width:100%;␣!important␣}␣</style>'))
```

4.1 Beispiel-Implementierung: Lokale Suchmaschine

4.1.1 Ziel der Beispiel-Implementierung

Im Folgenden wird eine Anwendung der zuvor theoretisch diskutierten Inhalte vorgestellt. Dabei soll eine lokale Suchmaschine entwickelt werden, 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. Um den Index zu speichern, wird die von Python mitgelieferte Datenstruktur "dictionary", welche im Grunde eine Hashmap ist, genutzt. Weiter werden einige Bibliotheken eingesetzt, welche einige Vorarbeit leisten und damit den Code der Beispiel-Implementierung auf das Wesentliche beschränken. So soll die grundlegende Arbeitsweise eines Information Retrieval-Systems dargelegt werden.

4.1.2 Genutzte Bibliotheken

Bevor mit der eigentlichen Implementierung der lokalen Suchmaschine begonnen werden kann, müssen einige Bibliotheken eingebunden werden. Darunter fallen Apache Tika, das Math-Modul von Python, os (um auf die Directories zugreifen zu können), python-magic, regular expressions (re) (und noch weitere, bei Bedarf einfügen!).

Tika

Tika liefert eine Parser, mit dessen Hilfe der Text aus - unter anderem - pdf-Dateien extrahiert werden kann. Mit dem Aufruf *parser.from_file(file)* kann eine pdf-Datei in reinen Text umgewandelt werden. Die Funktion liefert ein Dictionary zurück, welches einen Key *content* besitzt, über den auf den Inhalt der pdf-Datei zugegriffen werden kann.

python-magic

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

nltk

Die Bibliothek nltk (natural language toolkit) wird verwendet, um die Eingabetexte der Dokumente und die Eingaben des Nutzers zu normalisieren. Zudem wird Stemming mithilfe von nltk durchgeführt, um Wörter auf ihren Wortstamm zurückzuführen. In den unteren Methoden wird näheres über die genutzten Operationen erläutert.

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

4.1.3 Die Document-Klasse

Das Speichern er für das Retrieval wichtigen Informationen, geschieht mittels einer Document-Klasse. Diese Klasse hält alle Attribute, die wichtig sind, um das TF-IDF-Maß berechnen

zu können. Diese Attribute sind: - url - length - id

Die Variable *url* ist ein String und enthält den Pfad zum Dokument, welches durch das entsprechende Document-Objekt repräsentiert wird. *length* ist ein Integer und beinhaltet die Anzahl der Wörter, die in dem Dokument vorkommen und *id* ist die eindeutige Dokumenten-ID, zu der weiter unten noch genaueres gesagt wird.

```
class Document:
    def __init__(self, url, length, id):
        self.url = url
        self.length = length
        self.id = id
        self.score = {}
```

4.1.4 Der Index

Nachdem die benötigten Bibliotheken bekannt sind, kann der Index implementiert werden. Bevor dieser jedoch aufgebaut werden kann, sind einige Vorarbeiten nötig, die durch die vorgestellten Bibliotheken gestützt werden. Der Index wird im Folgenden als Klasse implementiert. Diese beinhaltet die folgenden Methoden, die in den folgenden Abschnitten genauer diskutiert werden: - buildIndex() - retrieve() - calcTFIDF()

Weiter werden die folgenden Member-Variablen benötigt: - hashmap - fileCount - docHashmap

Die Member-Variable *hashmap* ordnet allen Termen eine Menge von eindeutigen Dokumenten-IDs zu, in denen sie vorkommen. Im Dictionary *docHashmap* werden die Dokuemnten-IDs als Key genutzt, um eine Zurodnung von Dokumenten-IDs auf Document-Objekte zu ermöglichen. Die Variable *fileCount* ist ein Integer und wird für jedes gefundene Dokument um *1* hochgezählt. Damit ist diese Variable qualifiziert als eindeutige Dokumenten-ID zu fungieren, wofür sie genutzt wird.

```
class Index:
    hashmap = {} #dictionary
    fileCount = 0 #integer, Gesamtzahl aller gefunden Dateien
    docHashmap = {}
```

4.1.5 buildIndex

Die Methode *buildIndex* baut - wie der Name bereits vermuten lässt - den Index auf. Dabei dient ein Dictionary als Basis-Datenstruktur.

Der erste Schritt stellt das Iterieren über alle Directories dar. Gestartet wird bei Linux-Systemen im Root-Directory, unter Windows-Systemen muss über jede Partition iteriert werden. Als nächstes wird über alle Dateien in den Verzeichnissen iteriert. Für jede Datei wird durch python-magic ermittelt, ob es sich um ein pdf-Dokument handelt. Ist ein Dokument vom Typ *pdf*, wird mithilfe von tika der Text aus dem pdf-Dokument extrahiert.

Für jede entdeckte pdf-Datei wird ein Zähler erhöht, welcher eine eindeutige Dokumenten-ID darstellt. Anschließend wird mittels der Hilfsmethode *__processText* der Text der pdf-Dateien normalisiert. Diese Methode wird weiter unten genauer betrachtet.

Die letzten Schritte beinhalten das Anlegen eines neuen Dokumenten-Objekts, welches in das Dictionary *docHashmap* eingefügt wird. Zudem wird die Dokumenten-ID mithilfe der Hilfsmethode *__addToIndex* dem Dictionary *hasmap* hinzugefügt, welches den eigentlichen Index enthält.

```
def buildIndex(self):
    # alle Start-Verzeichnisse holen
    #start = self._getStartDirectories()
    #start = ["F:/Jonas/Uni"]
    start = ["C:/Users/marle/OneDrive/Studium"]
    # Magic-Instanz erstellen, um Datei-Typ bestimmen zu können
    mime = magic.Magic(mime=True)

    for s in start:
        for root, _dir, files in os.walk(s):
            for f in files:
                path = os.path.abspath(os.path.join(root, f))
                try:
                    if mime.from_file(path) == "application/pdf":
                        #print("pdf")
                    # in Text umwawndeln und tokenization durchführen
                        #print(path)
                        fileData = parser.from_file(path)
```

```
                        rawText = fileData['content']
                        self.fileCount += 1

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

    return

Index.buildIndex = buildIndex
```

Hilfsmethoden

In diesem Abschnitt werden die genutzten Hilfsmethoden kurz vorgestellt. Diese werden jedoch nicht in der Tiefe behandelt, wie die drei Haupt-Methoden behandelt werden.

```
def _getStartDirectories(self):
    start = []

    if platform.system() == "Linux":
        start.append("/")
    elif platform.system() == "Windows":
        start = ['%s:\\' % d for d in string.ascii_uppercase if os.path.exists('%s:' % d
    else:
        raise EnvironmentError

    return start

Index._getStartDirectories = _getStartDirectories
```

_addToIndex Diese Methode bekommt als Argumente eine Liste von Termen, die in einem pdf-Dokument vorkommen. Zudem wird eine eindeutige Dokumenten-ID übergeben.

Für jeden Term in *terms* wird versucht, eine Menge mit Dokumenten-IDs aus dem Index zu holen. Existiert bereits eine Menge, wird kein Fehler geworfen.

Schlägt der Versuch, die Menge für den Term *term* aus dem Dictionary zu holen, fehl, existiert noch keine Menge. In diesem Fall wird eine neue Menge erstellt und für den Term *term* ein Eintrag im Dictionary hinzugefügt, der auf die neu erstellte Menge referenziert.

```
def _addToIndex(self, documentID, terms):
    for t in terms:
        try:
            docs = self.hashmap[t]
            docs.add(documentID)
            self.hashmap.update({t : docs})
        except KeyError:
            docs = {documentID}
            self.hashmap.update({t : docs})

Index._addToIndex = _addToIndex
```

_preprocessText Diese Methode dient der Vorverarbeitung der Texte, die in den pdf-Dokumenten stehen. Als erster Schritt wird der gesamte Text in Lower-Case gesetzt, damit bei der Suche später die Groß- bzw. Kleinschreibung unwichtig ist. Im nächsten Schritt werden alle Zahlen aus dem Text entfernt. Als nächstes wird mithilfe der Klasse *RegexpTokenizer*, die die nltk-Bibliothek mitliefert, der String *txt* in eine Liste von Tokens aufgespalten. Was als Token gewertet wird, wird mithilfe einer *regular expression* definiert, die dem Konstruktor des *RegexpTokenizer* mitgegeben. Mithilfe der *tokenize*-Methode wird die *regular expression* auf den übergebenen String angewendet. Jeder Substring des mitgegebenen Arguments, der die *regular expression* erfüllt, wird einer Liste angefügt. Diese Liste enthält am Ende der Verarbeitung alle Substrings von *txt*, die nur Buchstaben enthalten oder nur Buchstaben enthalten und mit einem Bindestrich (-) enden.

Der Bindestrich ist wichtig, da mit dessen Hilfe, alle Wörter gefunden werden können, die

im Text aufgrund eines Zeilenumbruchs getrennt wurden. Diese Wörter werden innerhlab der for-Schleife zusammengefügt und der Bindestrich wird entfernt.

Diese Methode ist jedoch nicht immer korrekt, denn es kann auch folgender Fall eintreten: Eine Zeile endet mit z.B. Damen-, die nächste Zeile geht mit z.B. und Herrenschuhe weiter. In diesem Fall ist der Bindestrich gewollt, der Algorithmus fügt jedoch die Wörter *Damen* und *und* zu einem Wort zusammen. Da dieser Fall jedoch sehr selten auftritt, wird in Kauf genommen, dass ab und zu die Wörter falsch zusammengefügt werden.

```
def _preprocessText(self, txt):
    # lower all:
    txt = txt.lower()

    # remove digits
    txt = re.sub(r'\d+', '', txt)

    # tokenize the text
    tokenizer = RegexpTokenizer(r'[a-zA-Z]+-$|\w+')
    result = tokenizer.tokenize(txt)

    # concatenate divided words
    for word in result:
        if word[-1] == '-':
            ind = result.index(word)
            corrected = word[:-1]+result[ind+1]
            result[ind] = corrected
            del result[ind+1]
    return result

Index._preprocessText = _preprocessText
```

retrieve Die retrieve-Methode dient der Suche. Die Idee dabei ist, dass der Nutzer ein oder mehrere Schlagworte eingeben kann, auf deren Basis die am besten passenden Dokumente zurückgeliefert werden. Auch die Schlagworte werden den gleichen Normalisierungs-Prozess durchlaufen wie die Texte der Dokumente.

Zunächst wird der Such-String des Nutzers mittels der bereits bekannten Methode *__prepro-*

cessText normalisiert. Im zweiten Schritt wird eine leere Menge angelegt, in der die Dokumenten-IDs, die zu den Termen gefunden werden, gespeichert. Mithilfe der for-Schleife wird über *processedStrings* iteriert. Für jedes Wort wird versucht, die Menge aller Dokumenten-IDs zu dem Term *word* aus dem Index zu beschaffen. Existiert die Menge zu dem Term *word*, wird die Vereinigung der bereits in der Menge *result* stehenden Dokumenten-IDs und der mit dem Term *word* gefundenen Dokumenten-IDs gebildet. Existiert der Term *word* nicht als Key im Index, wird beim nächsten Term der Liste *processedStrings* fortgefahren.

```
def retrieve(self, searchString):
    # pre-processing
    processedStrings = self._preprocessText(searchString)
    result = set()
    for word in processedStrings:
        try:
            documents = set(self.hashmap[word])
            result = result.union(documents)
        except KeyError:
            continue
    return result

Index.retrieve = retrieve

ind = Index()
ind.buildIndex()
```

4.2 Probleme

- er findet *.dat-Dateien
- Kontext der Wörter nicht einbezogen
- Vereinigung der Ergebnismengen sorgen dafür, dass Ergebnisse für $t_1 \cup t_2 \cup .. \cup t_n$ zurückgegeben werden -> kann durch Gewichtung verbessert/umgangen werden
- kein Stemming bisher

- Queries mit einem Buchstaben finden viele Dokumente (vor allem Formeln etc.) -> durch Stemming gelöst, falls nicht gewollt
- Index.build() braucht sehr lange -> parallelisieren?
- Index muss jedes Mal neu aufgebaut werden -> via Pickle, JSON oder XML speichern und ziehen bei Start des Programms

```
def tf_idf(self, searchString):
    tfDict = {}
    ind = Index()
    termList = ind._preprocessText(searchString)
    for term in termList:
        tfDict[term] = 0

    fileData = parser.from_file(self.url)
    rawText = fileData['content']
    rawTextList = ind._preprocessText(rawText)

    for term in rawTextList:
        if term in termList:
            tfDict[term] = tfDict[term]+1

    self.score = tfDict

Document.tf_idf = tf_idf

resultSet = ind.retrieve("information␣retrieval")
#for result in resultSet:
    #print(ind.docHashmap[result].url)
for document in resultSet:
    doc = ind.docHashmap[document]
    doc.tf_idf("information␣retrieval")
    print(doc.score)

{'information': 16, 'retrieval': 1}
{'information': 1, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 0, 'retrieval': 1}
```

```
{'information': 4, 'retrieval': 0}
{'information': 0, 'retrieval': 5}
{'information': 6, 'retrieval': 0}
{'information': 4, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 5, 'retrieval': 0}
{'information': 13, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 8, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 2, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 444, 'retrieval': 54}
{'information': 2, 'retrieval': 0}
{'information': 5, 'retrieval': 0}
{'information': 8, 'retrieval': 0}
{'information': 2, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 2, 'retrieval': 0}
{'information': 2, 'retrieval': 0}
{'information': 5, 'retrieval': 0}
{'information': 3, 'retrieval': 0}
{'information': 2, 'retrieval': 0}
{'information': 6, 'retrieval': 0}
{'information': 6, 'retrieval': 0}
{'information': 3, 'retrieval': 0}
{'information': 5, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 3, 'retrieval': 0}
{'information': 6, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 2, 'retrieval': 0}
{'information': 15, 'retrieval': 1}
{'information': 3, 'retrieval': 0}
{'information': 15, 'retrieval': 0}
{'information': 16, 'retrieval': 0}
```

{'information': 1, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 2, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 3, 'retrieval': 0}
{'information': 595, 'retrieval': 711}
{'information': 4, 'retrieval': 0}
{'information': 2, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 2, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 2, 'retrieval': 0}
{'information': 2, 'retrieval': 0}
{'information': 2, 'retrieval': 0}
{'information': 2, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 1, 'retrieval': 0}
{'information': 154, 'retrieval': 1}
{'information': 1, 'retrieval': 0}
{'information': 1, 'retrieval': 0}

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

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