Elementares Tokenizing, Indexing, und die Implementierung von vektorraumbasiertem Retrieval

Viele Folien in diesem Abschnitt sind eine deutsche Übersetzung der Folien von Raymond J. Mooney (http://www.cs.utexas.edu/users/mooney/ir-course/).

KSM Implementierung

- KSM wird ein einfaches in Java geschriebenes Vektorraum-Retrieval-System werden
- Entsteht während der praktischen Übung bei jedem Teilnehmer
- Wird mit HTML und ASCII Dateien umgehen können und einen einfachen Spider enthalten

Einfaches Tokenizing

- Zerlege Text in eine Sequenz einzelner Token (Terme).
- Manchmal können Interpunktion (e-mail), Zahlen (1999), und Groß-/Kleinschreibung (Republican vs. republican) ein aussagekräftiger Teil eines Token sein.
- Häufig sind sie es jedoch nicht.
- Die einfachste Annäherung ist, alle Zahlen und Interpunktionen zu ignorieren und nur ununterbrochene Strings alphabetischer Zeichen ohne Berücksichtigung der Groß- und Kleinschreibung als Token zu verwenden.

Tokenizing HTML

- Sollte Text in HTML Befehlen, der typischerweise nicht vom Anwender gesehen werden kann, als Token im Modell enthalten sein?
  - Worte, die in URLs erscheinen.
  - Worte, die in “Metatext” von Bildern erscheinen.
- Einfachste in KSM verwendete Annäherung ist, alle HTML-Tag-Informationen (zwischen “<“ und “>”) beim Berechnen der Tokens auszuschließen.

Dokumente in KSM

- Dokumente aus verschiedenen Quellen
  - ASCII-Datei
  - HTML-Datei bzw. URL
  - String
- Auch Anfragen sind Dokumente!
  - String

Stopwörter

- Es ist typisch, Wörter mit hoher Häufigkeit *auszuschließen* (z.B. Funktionswörter: “a”, “the”, “in”, “to”; Pronomen: “I”, “he”, “she”, “it”).
- Stopwörter sind sprachabhängig. KSM verwendet für Englisch eine Standardmenge von etwa 500.
- Aus Effizienzgründen sollte man Stopwörter als Strings in einer Hash-Tabelle abspeichern, um auf diese in konstanter Zeit zugreifen zu können.

Stopwortlisten für verschiedene Sprachen findet man z.B. unter: http://www.unine.ch/info/clef/

Stemming

- Reduziere Token auf die “Stamm”-Form eines Wortes, um morphologische Variationen zu erkennen.
  - “computer”, “computational”, “computation” werden alle auf den gleichen Wortstamm reduziert
- Korrekte morphologische Analyse ist sprach-spezifisch und kann komplex sein (meist wörterbuchbasiert).
- Stemming löscht in einer iterativen Art und Weise relativ “blindlings” bekannte Affixe (Präfixe und Suffixe).

Porter Stemmer

- Einfaches Verfahren für das Entfernen von Suffixen im Englischen.
- Ist ohne ein Lexikon verwendbar.
- Kann ungewöhnliche Stämme bilden, die weder englische Worte noch Wortstämme im grammatikalischen Sinne sind:
  - “computer”, “computational”, “computation” alle reduziert auf den gleichen Token “comput”
- Kann eigenständige Worte zu demselben Stamm verschmelzen (auf das gleiche Token reduzieren)
- Erkennt nicht alle morphologischen Abstammungen

Porter Stemmer Algorithmus

- Der Algorithmus besteht aus einer Kaskade von Substitutionen für gegebene Bedingungen
  - GENERALIZATIONS
  - GENERALIZATION
  - GENERALIZE
  - GENERAL
  - GENER
- Onlineversion:
  http://maya.cs.depaul.edu/~classes/ds575/porter.html

Porter, M.F., 1980, An algorithm for suffix stripping, Program, 14(3) :130-137

Folgende Folien stammen von Prof. Bonnie J. Dorr

Porter Stemmer Algorithm

***<S>** = ends with <S>
v = contains a V (Vokal)
***d** = ends with double C (Konsonant)
***o** = ends with CVC second C is not W, X or Y

Step 1: Plural Nouns and Third Person Singular Verbs

SSES → SS		caresses	→ caress
IES → I		ponies	→ poni
		ties	→ ti
SS → SS		caress	→ caress
S →		cats	→ cat

Step 2a: Verbal Past Tense and Progressive Forms

	(m > 0) EED → EE	feed → feed, agreed → agree
i	(*v*) ED →	plastered → plaster, bled → bled
ii	(*v*) ING →	motoring → motor, sing → sing

Step 2b: If 2a.i or 2a.ii is successful, Cleanup

AT → ATE	conflat(ed) → conflate
BL → BLE	troubl(ed) → trouble
IZ → IZE	siz(ed) → size
(*d and not (*L or *S or *Z)) → single letter	hopp(ing) → hop, tann(ed) → tan
	hiss(ing) → hiss, fizz(ed) → fizz
(M=1 and *o) → E	fail(ing) → fail, fil(ing) → file

Porter Stemmer Algorithm

***<S>** = ends with <S>
v = contains a V
***d** = ends with double C
***o** = ends with CVC second C is not W, X or Y

Step 3: Y → I

(*v*) Y → I	happy → happi
	sky → sky

Step 4: Derivational Morphology I: Multiple Suffixes

(m>0) ATIONAL	-> ATE	relational	->	relate
(m>0) TIONAL	-> TION	conditional	->	condition
		rational	->	rational
(m>0) ENCI	-> ENCE	valenci	->	valence
(m>0) ANCI	-> ANCE	hesitanci	->	hesitance
(m>0) IZER	-> IZE	digitizer	->	digitize
(m>0) ABLI	-> ABLE	conformabli	->	conformable
(m>0) ALLI	-> AL	radicalli	->	radical
(m>0) ENTLI	-> ENT	differentli	->	different
(m>0) ELI	-> E	vileli	- >	vile
(m>0) OUSLI	-> OUS	analogousli	->	analogous
(m>0) IZATION	-> IZE	vietnamization	->	vietnamize
(m>0) ATION	-> ATE	predication	->	predicate
(m>0) ATOR	-> ATE	operator	->	operate
(m>0) ALISM	-> AL	feudalism	->	feudal
(m>0) IVENESS	-> IVE	decisiveness	->	decisive
(m>0) FULNESS	-> FUL	hopefulness	->	hopeful
(m>0) OUSNESS	-> OUS	callousness	->	callous
(m>0) ALITI	-> AL	formaliti	->	formal
(m>0) IVITI	-> IVE	sensitiviti	->	sensitive
(m>0) BILITI	-> BLE	sensibiliti	->	sensible

Porter Stemmer Algorithm

***<S>** = ends with <S>
v = contains a V
***d** = ends with double C
***o** = ends with CVC second C is not W, X or Y

Step 7a: Cleanup

(m>1) E →	probate → probat
	rate → rate
(m=1 and not *o) E →	cease → ceas

Step 7b: More Cleanup

(m > 1 and *d and *L) → single letter	controll → control
	roll → roll

Fehler des Porter Stemmer

- "over-stemming", zuviel wurde entfernt:
  - organization, organ → organ
  - police, policy → polic
  - arm, army → arm
- "under-stemming", zu wenig entfernt:
  - cylinder (cylind), cylindrical (cylindr)
  - Create (creat), creation
  - Europe (europ), European

Dünn besetzte (sparse) Vektoren

- Das Vokabular, und damit auch die Dimensionalität des Vektorraums, kann sehr groß werden, ~10^4 Terme.
- Jedoch enthalten die meisten Dokumente und Anfragen nur sehr wenige Wörter, somit sind Vektoren dünn besetzt („sparse"), d.h. die meisten Einträge sind 0.
- Man benötigt effiziente Methoden zur Speicherung und zum Rechnen mit dünn besetzten Vektoren.

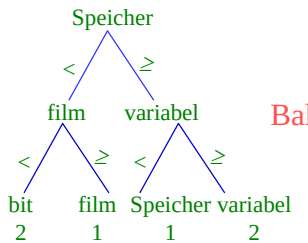
Sparse Vektoren als Listen

- Idee: Speichere nur Tokens, deren Gewicht ungleich 0 ist, zusammen mit ihrem Gewicht, als Vektoren in einer verlinkten Liste.
  - Platzbedarf proportional zur Anzahl der Tokens (n) im Dokument
  - Erfordert eine lineare Suche in der Liste aller Tokens, um das Gewicht eines spezifischen Token zu finden (oder zu verändern).
  - Erfordert im schlimmsten Fall quadratischen Zeitaufwand, um den Vektor für ein Dokument zu berechnen:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

Sparse Vektoren als Bäume

- Indexiere Tokens eines Dokumentes in einem balancierten binären Baum oder einem Trie (zeichenweiser Schlüsselvergleich), bei dem die Gewichte der Tokens an den Blättern gespeichert sind.



Balancierter binärer Baum

Sparse Vektoren als Bäume (cont.)

- Overhead beim Speichern der Baumstruktur: ~2*n* Knoten.
- Zeit: O(log *n*) um das Gewicht eines spezifischen Tokens zu finden oder zu aktualisieren.
- Zeit: O(*n* log *n*) um den Vektor zu erzeugen.

Sparse Vektoren als Hash-Tabellen

- Speichere die Token in einer Hash-Tabelle, mit Token-String als Schlüssel und Gewicht als Wert.
  - Overhead beim Speichern in einer Hash-Tabelle ~1.5*n*.
  - Tabelle muss in Hauptspeicher passen.
  - Konstante Zeit, um das Gewicht eines spezifischen Tokens zu finden oder zu aktualisieren. (Kollisionen werden ignoriert).
  - Zeit um den Vektor zu erzeugen: O(*n*) (Kollisionen werden ignoriert).

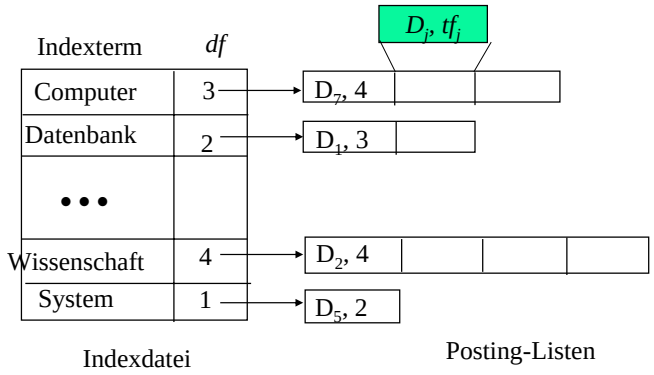
Sparse Vektor in KSM

- Hash-Tabelle um die Terme eines Dokumentes zu verwalten (_termCounts: String → Integer).
- _termCounts stellt die interne Datenstruktur der Dokumentklasse dar

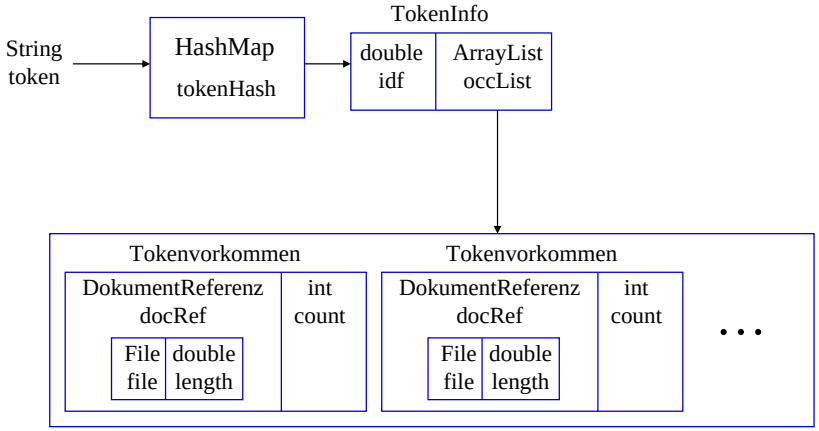
Implementierung basierend auf Invertierten Dateien

- In der Praxis werden Dokumentvektoren nicht direkt gespeichert; eine invertierte Organisation der Daten bietet eine deutlich höhere Effizienz.
- Der Keyword-to-document Index kann als Hash-Tabelle, sortiertes Array oder als Baumstruktur (Trie, Tree) gespeichert werden.
- Kritischer Punkt ist der Zugriff auf die Tokens in logarithmischer oder konstanter Zeit.

Invertierter Index



KSM Invertierter Index



Kreieren eines invertierten Indexes

Kreiere eine leere HashMap *H*;

Für jedes Dokument *D* (z.B. jede Datei in einem Input- Verzeichnis):

Kreiere einen HashMap-Vector *V* (_termCounts) für *D*;

Für jedes (nicht-null) Token *T* in *V*:

Wenn T nicht bereits in *H* ist, kreiere eine leere TokenInfo für *T* und füge diese in *H* ein;

Kreiere ein Tokenvorkommen für *T* in *D* und füge es zur *occList* in die TokenInfo für *T*;

Berechne IDF für alle Tokens in *H*;

Berechne Vektorlänge für alle Dokumente in *H*;

Berechnung IDF

N sei die Gesamtzahl aller Dokumente;

Für jedes Token *T* in *H*:

Bestimme die Gesamtzahl *M* der Dokumente, in denen *T* vorkommt (die Länge *occList* von *T*);

Setze den IDF-Wert für T auf log(N/M);

Beachte, dass dies einen zweiten Durchgang durch alle Tokens erfordert, nachdem alle Dokumente indexiert worden sind.

Vektorlänge der Dokumente

- Wdh. (aus der Linearen Algebra): Die Länge eines (Dokument-) Vektors ist die Quadratwurzel der Summe der Quadrate der Gewichte seiner Tokens.
- Das Gewicht eines Tokens ist hier:

  TF * IDF
- Daher muss gewartet werden, bis alle IDF-Werte bekannt sind (und demzufolge bis alle Dokumente indexiert wurden), bevor die Dokumentlänge bestimmt werden kann.

Berechnung der Dokumentlängen

Gehe davon aus, dass die Länge aller Dokumentvektoren mit 0.0 initialisiert werden;
Für jedes Token *T* in *H*:
  I sei das IDF-Gewicht von *T*;
  Für jedes Tokenvorkommen von *T* in Dokument *D*:
    C sei die Anzahl von *T* in *D*;
    Inkrementiere die Länge von *D* mit $(I*C)^2$;
Für jedes Dokument *D* in *H*:
  Setze die Länge von *D* als die Quadratwurzel der aktuell gespeicherten Länge;

Zeitkomplexität beim Indexieren

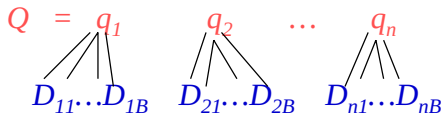
- Die Komplexität des Vektorerstellens und des Indexierens für ein Dokument mit n Tokens ist $O(n)$.
- Indexieren von m Dokumenten kostet $O(m\ n)$.
- Berechnung der IDF-Werte für jedes Token im Vokabular V kostet $O(|V|)$.
- Aufwand für die Berechnung der einzelnen Vektorlängen beträgt ebenfalls $O(m\ n)$.
- Wegen $|V| \leq m\ n$ beträgt der Zeitaufwand für den kompletten Prozess $O(m\ n)$, was auch der Komplexität zum Einlesen des Korpus entspricht.

Retrieval mit invertiertem Index

- Tokens, die weder in der Anfrage noch im Dokument vorkommen, haben keinen Einfluss auf die Kosinus-Ähnlichkeit.
  - Das Produkt dieser Tokengewichte ist null und trägt daher nicht zum Skalarprodukt bei.
- Normalerweise ist die Anfrage ziemlich kurz und demzufolge ihr Vektor *äußerst* sparse.
- Verwende den invertierten Index, um die kleine Menge von Dokumenten zu finden, die zumindest eines der Anfragewörter enthalten.

Effizienz von invertierten Anfragen

- Angenommen, ein Anfragewort erscheint im Durchschnitt in *B D*okumenten:



- Dann beträgt die Retrievalzeit O(|*Q*| *B*) und ist damit im allgemeinen **viel** besser als das naive Retrieval mit O(|*V*| *N*), das alle *N* Dokumente überprüft, da |*Q*| << |*V*| und *B* << *N*.

Verarbeitung einer Anfrage

- Berechne die Kosinus-Ähnlichkeit eines jedes indexierten Dokumentes inkrementell, indem die Anfrageworte nacheinander abgearbeitet werden.
- Um einen Gesamtscore jedes Dokuments zu ermitteln, speichert man die gefundenen Dokumente in einer Hash-Tabelle. Die Referenz auf das Dokument wird der Schlüssel und der bisher bestimmte Score der Wert.

Algorithmus: Anfrage gegen Invertierten Index

Kreiere einen HashMap-Vektor *Q* für die Anfrage.

Kreiere leere HashMap *R*, um gefundene Dokumente und deren Werte zu speichern.

Für jedes Token *T* in *Q*:

  Sei *I* das IDF von *T* und *K* die Anzahl von *T* in *Q*;

  Setze das Gewicht von *T* in *Q*: $W = K * I$;

  Sei L die Liste der Tokenvorkommen von *T* in *H*;

  Für jedes Token *O* in *L*:

    Sei *D* das Dokument zu *O* und *C* die Anzahl von *T* in *O*
    (*tf* von *T* in *D*);

    Wenn *D* nicht bereits in *R* ist (*D* wurde zuvor nicht gefunden)

      dann füge *D* zu *R* und initialisiere Wert auf 0.0;

    Erhöhe den Wert von *D* um $W * I * C$;
    (Produkt des Gewichtes von *T* in *Q* und *D*)

Retrieval-Algorithmus (fort.)

Berechne die Länge *L* des Vektors *Q* (Quadratwurzel der Summe der Quadrate seiner Gewichte).

Für jedes gewonnene Dokument *D* in *R*:

  Sei S der aktuellen Score von *D*;

    (*S* ist das Skalarprodukt von *D* und *Q*)

  Sei *Y* die Länge von *D*, wie es in der Dokumentenreferenz gespeichert ist;

  Normalisiere den endgültigen Score von *D* durch $S/(L * Y)$;

Sortiere die gewonnenen Dokumente in *R* anhand des Scores und lege das Ergebnisse in einem Array ab.

Effizienzhinweis

- Um die Berechnungseffizienz zu steigern und eine zusätzliche Iteration durch die Tokens in der Anfrage zu vermeiden, wird die Berechnung der Länge des Anfragevektors in die Verarbeitung der Anfragetoken integriert.

Anwenderschnittstelle

Bis der Anwender mit einer leeren Anfrage abschließt:

Fordere den Anwender auf, eine Anfrage *Q* zu stellen.

Berechne die geordnete Liste *R* der gefundenen *D* für *Q*;

Drucke die Namen der ersten *N* Dokumente in *R*;

Bis der Anwender mit einem leeren Befehl abschließt:

Fordere den Anwender auf, einen der folgenden Befehle als Ergebnis dieser Anfrage einzugeben:

1) Zeige die nächsten *N* Elemente der Liste *R*;

2) Zeige das Mte gefundene Dokument;

(Dokument wird im Browser-Fenster gezeigt)

Effizientes Erstellen eines invertierten Indexes für sehr große Datenmengen

Aus: Managing Gigabytes: Compressing and Indexing Documents and Images, Ian H. Witten, Alistair Moffat, and Timothy C. Bell, 1999

Beispieldatensatz

- 5 GB Datensatz
- 5 Mill. Dokumente
- 1 Mill. unterschiedliche Worte
- 800 Mill. Worte insgesamt
- 400 Mill. index pointers
- 30 MB für das Lexikon
- 400 MB für den komprimierten Index

Aus: Managing Gigabytes: Compressing and Indexing Documents and Images, Ian H. Witten, Alistair Moffat, and Timothy C. Bell, 1999

Hauptspeicher-basierter Invertierter Index

- Hashtabellen-basierter Ansatz mit einer Linkliste ist eine der effizientesten Varianten.
- Angenommen, man liest die Daten mit 2Mb/s von der Platte, dann braucht man 40 min für 5GB
- Verarbeiten (tokenizing, stemming, etc.) ca. 4 h
- Schreiben des invertierten Indexes ca. 40 min
- Bei 10 bytes für jeden Knoten und 400 Mill. Knoten braucht man 4GB Hauptspeicher.

Linked-Liste auf der Festplatte

- Idee: Linked-Liste der Dokumentnummern auf Platte speichern
  - Erste Schritte des Algorithmus sind weiter effizient
  - Nach dem Aufbau der Linked-Liste muss diese zum Schreiben des Invertierten Indexes in Termordnung durchlaufen werden.
  - 10 ms für jeden Zugriff auf die Platte um 10 byte zu lesen
  - Bei 400 Mill Einträgen führt dies zu 4 Mill Sekunden oder 6 Wochen.

Sortierter Index auf Platte

- Folgende wesentliche Schritte umfasst der Algorithmus:
  - Initialisiere Datenstrukturen im Hauptspeicher
  - Lese Texte von Platte, verarbeite Dokumente und Schreibe sequentiell in ein tmp-File für jeden Term eines Dokumentes einen Datensatz $<t, d, f_{d,t}>$
  - Sortiere das tmp-File nach Termen, um das invertierte File daraus zu erzeugen
  - Schreibe invertiertes File

Sortierter Index auf Platte

- Initialisiere Datenstrukturen im Hauptspeicher
  - Erstelle eine leere Wörterbuchstruktur *S*
  - Erstelle eine leeres tmp-File auf der Festplatte

Sortierter Index auf Platte

- tmp-File für jeden Term eines Dokumentes
  - Für jedes Dokument *d*
    - Lese und parse das Dokument *d*
    - Für jeden Term *t* aus Dokument *d*
      - Bestimme die Häufigkeit $f_{d,t}$ für Term *t* aus Dokument *d*
      - Suche nach *t* in *S*; falls *t* nicht in *S* füge *t* hinzu
      - Schreibe den Datensatz $<t, d, f_{d,t}>$ in das tmp-File wobei *t* durch seine Termnummer in S repräsentiert wird.

Sortierter Index auf Platte

- Sortieren des tmp-Files
  - Sei *k* die Anzahl an Datensätzen, die der Hauptspeicher aufnehmen kann
    - Lese *k* Datensätze vom tmp-File
    - Sortiere diese Datensätze aufsteigend nach *t* und *d*
    - Schreibe den sortierten Teil zurück ins tmp-File
    - Wiederhole dies, bis keine Datensätze verbleiben
  - Paarweises Mergen der *k* Durchläufe des tmp-Files, bis alle Datensätze sortiert sind

Sortierter Index auf Platte

- Schreibe invertiertes File
  - Für jeden Term *t*
    - Beginne einen neuen Eintrag im invertierten File
    - Lese alle Datensätze $<t, d, f_{d,t}>$ zu Term *t* aus dem tmp-File, und erzeuge eine Eintrag für Term *t*
    - Wenn nötig, komprimiere diesen Eintrag
    - Hänge den Eintrag an das invertierte File auf der Festplatte an

Sortierter Index auf Platte; Aufwand?

- Bei 40 MB Hauptspeicher braucht man ca. 20 Stunden und 8 GB Speicher zusätzlichen Plattenplatz
- 5 Stunden, um Dokumente zu verarbeiten und das tmp-File zu erstellen
- Bei 40MB Hauptspeicher werden k = 100 Blöcke gebildet und in 4 Stunden sortiert
- „Mergen“ dieser 100 Blöcke in 7 Durchgängen dauert ca. 9 Stunden
- Ca. 2 Stunden werden benötigt, um aus dem tmp-File den invertierten Index zu erstellen

Zwei weitere Methoden zur Steigerung der Effizienz

- Kompression der tmp-Datei
  - Kompression reduziert den Overhead durch das Schreiben des tmp-Files auf die Platte
  - Nur noch Abstände (Gaps) werden gespeichert
  - Sowohl die Dokumentenliste und deren Häufigkeit als auch die Liste der Terme können komprimiert werden
  - Bei 40MB Speicher, 680 MB Festplattenplatz, 26 Stunden
- Multiway merge
  - Nicht nur 2 sondern z.B. 4 aus *k*-Durchläufen des tmp-Files werden auf einmal gemerged
  - Dadurch reduziert sich die Anzahl der Durchläufe
  - Bei 40MB Speicher, 540 MB Festplattenplatz, 11 Stunden