Information Retrieval Models

Outline

- So far we have seen how text can be processed
- In the following we will see how, once text has been preprocessed, we can leverage it to build an information retrieval system
- In particular, we will learn
  - How a document is represented
  - How a query is matched against a document

Sources

- Most of the material in the following is taken from the “Modern Information Retrieval” book by Baeza-Yates & Ribeiro-Neto

IR Models

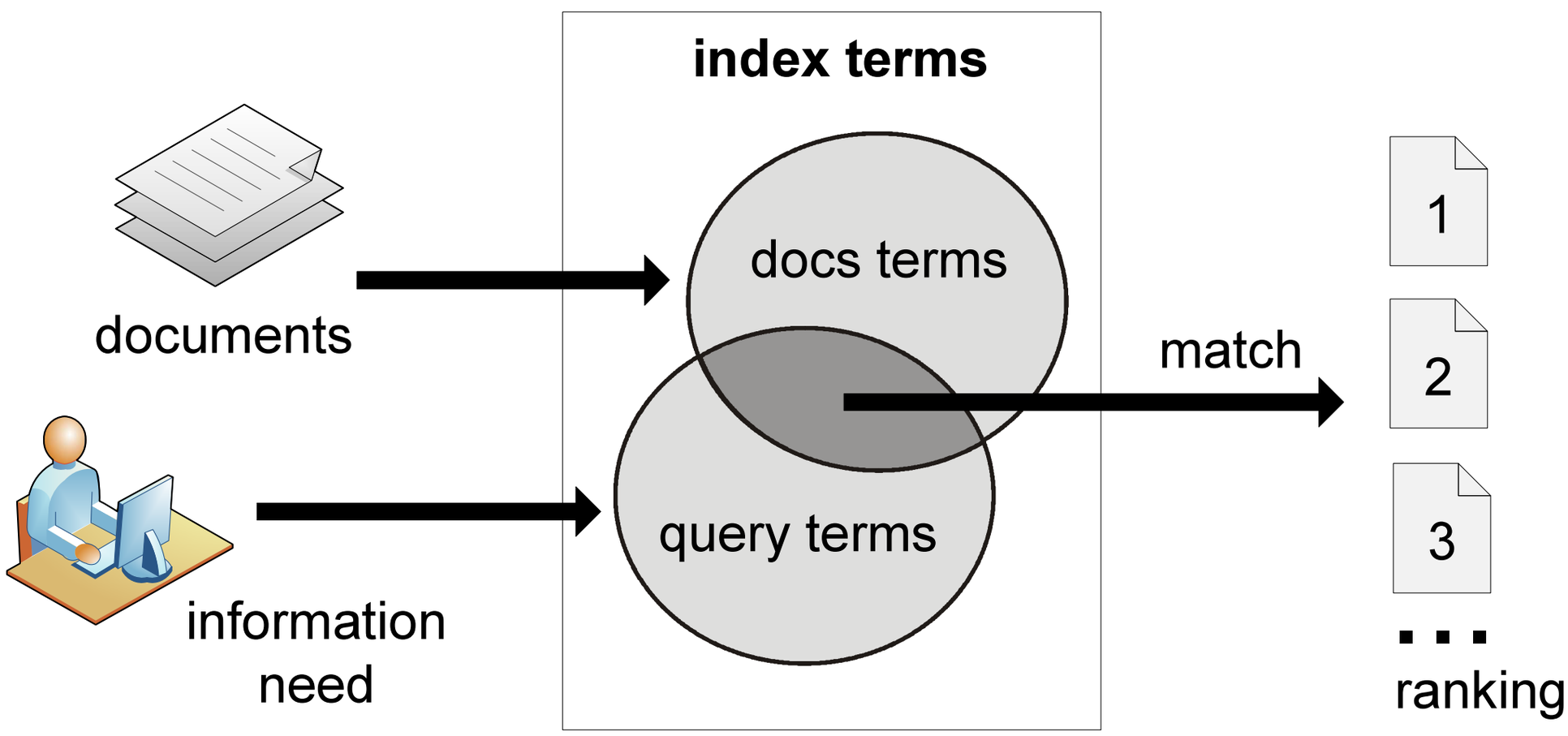
- **Modeling** in IR is a complex process aimed at producing a ranking function
  - **Ranking function**: a function that assigns scores to documents with regard to a given query
- This process consists of two main tasks:
  - The conception of a logical framework for representing documents and queries
  - The definition of a ranking function that allows quantifying the similarities among documents and queries

Modeling and Ranking

- IR systems usually adopt **index terms** to index and retrieve documents
- Index term:
  - In a restricted sense: it is a keyword that has some meaning on its own; usually plays the role of a noun
  - In a more general form: it is any word that appears in a document
- Retrieval based on index terms can be implemented efficiently
- Also, index terms are simple to refer to in a query
- Simplicity is important because it reduces the effort of query formulation

Introduction

- Information retrieval process

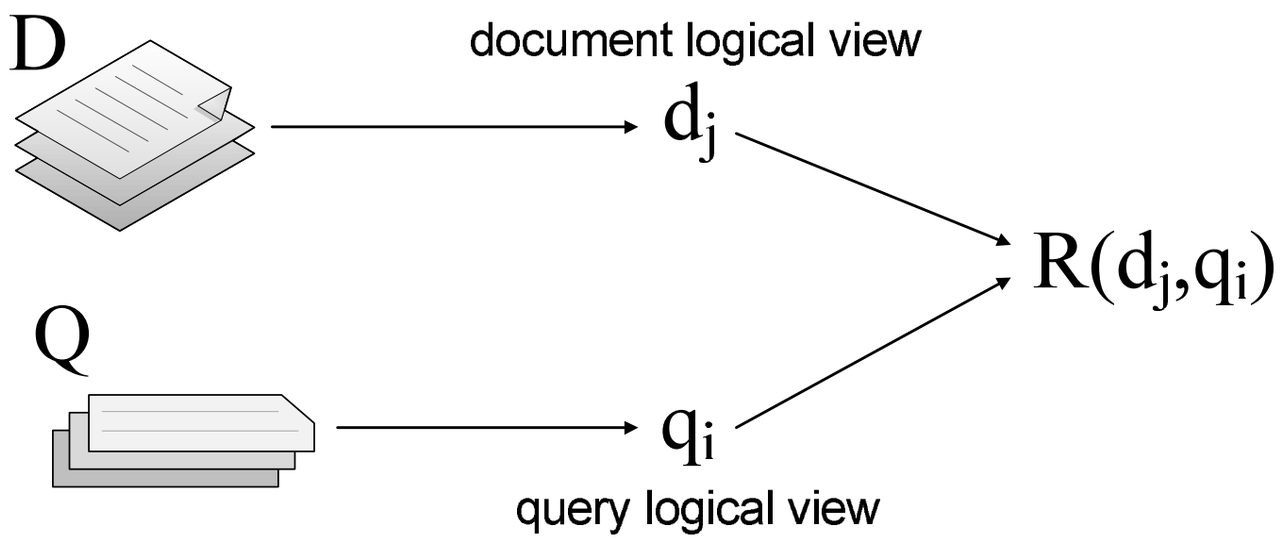


Introduction

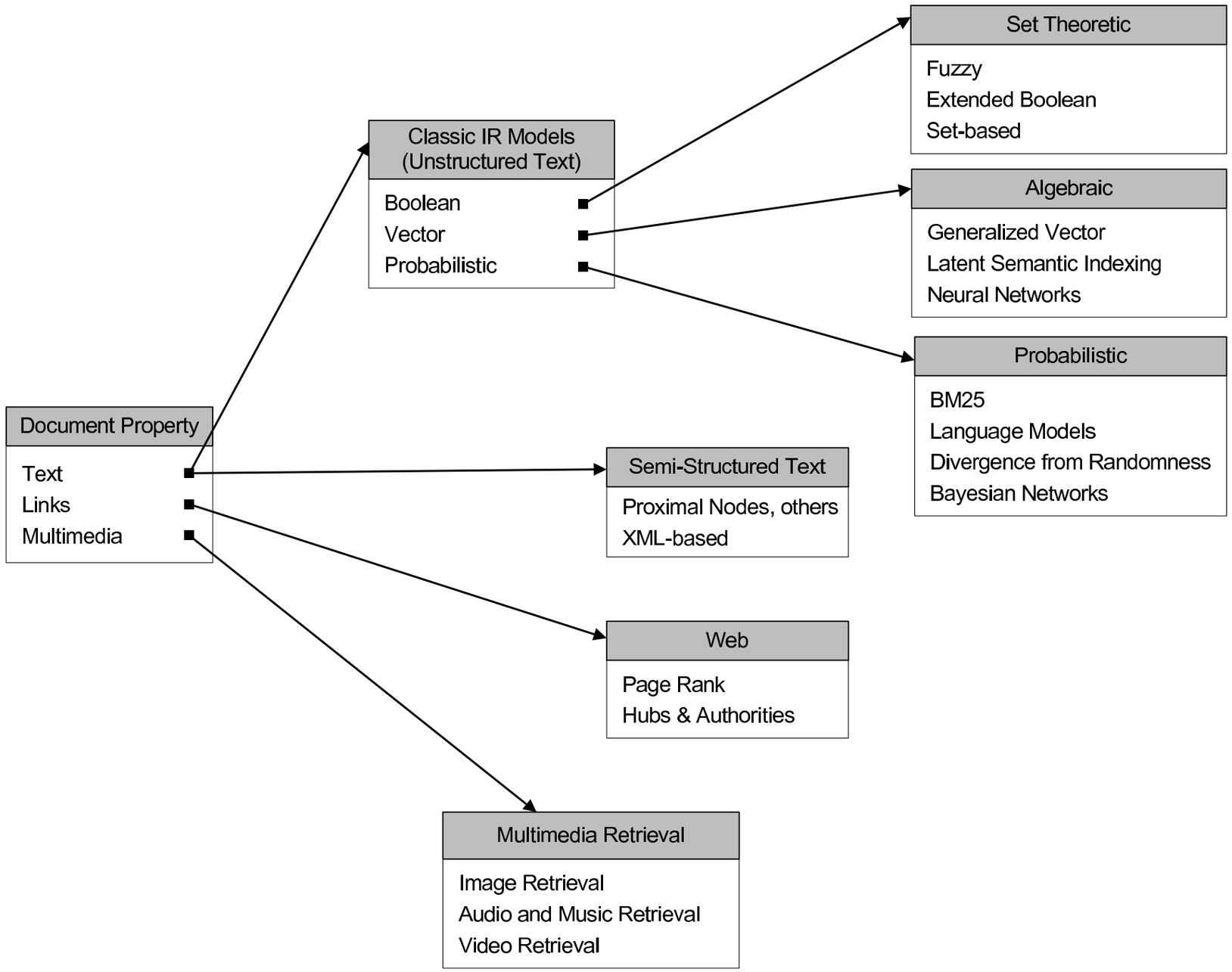
- A **ranking** is an ordering of the documents that (hopefully) reflects their **relevance** to a user query
- Thus, any IR system has to deal with the problem of predicting which documents the users will find relevant
- This problem naturally embodies a degree of uncertainty, or vagueness

IR Models

- An **IR model** is a quadruple $[\mathbf{D}, \mathbf{Q}, \mathcal{F}, R(q_i, d_j)]$ where
  1. $\mathbf{D}$ is a set of logical views for the documents in the collection
  2. $\mathbf{Q}$ is a set of logical views for the user queries
  3. $\mathcal{F}$ is a framework for modeling documents and queries
  4. $R(q_i, d_j)$ is a ranking function



A Taxonomy of IR Models



Basic Concepts

- Each document is represented by a set of representative keywords or index terms
- An index term is a word or group of consecutive words in a document
- A pre-selected set of index terms can be used to summarize the document contents
- However, it might be interesting to assume that all words are index terms (full text representation)

Basic Concepts

- Let,
  - t be the number of index terms in the document collection
  - k_i be a generic index term
- Then,
  - The **vocabulary** $V = \{k_1, \ldots, k_t\}$ is the set of all distinct index terms in the collection

$V =$ | k_1 k_2 k_3 ... k_t | vocabulary of t index terms

Basic Concepts

- Documents and queries can be represented by **patterns of term co-occurrences**

$V =$

k_1	k_2	k_3	...	k_t
1	0	0	...	0
⋮				
1	1	1	...	1

pattern that represents documents (and queries) with the term k_1 and no other

pattern that represents documents (and queries) with all index terms

- Each of these patterns of term co-occurence is called a **term conjunctive component**
- For each document d_j (or query q) we associate a unique term conjunctive component $c(d_j)$ (or $c(q)$)

The Term-Document Matrix

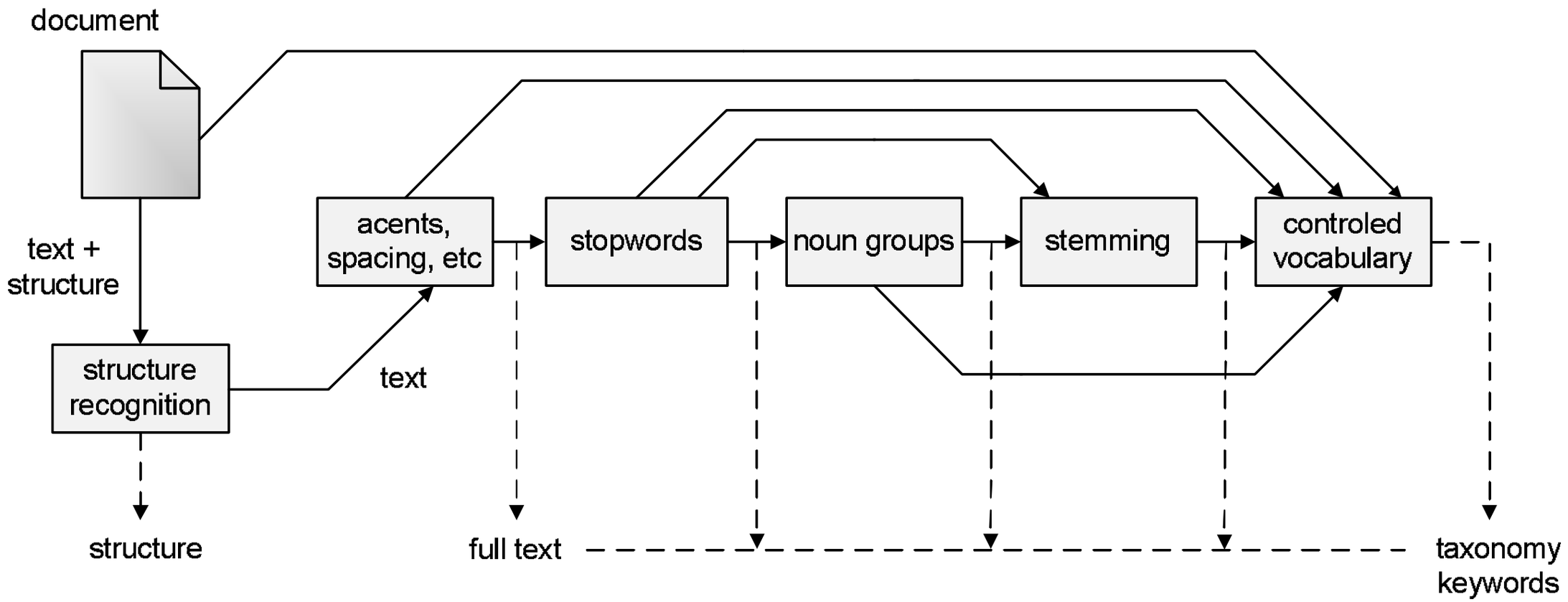
- The occurrence of a term k_i in a document d_j establishes a relation between k_i and d_j
- A **term-document relation** between k_i and d_j can be quantified by the frequency of the term in the document
- In matrix form, this can written as

$$\begin{array}{c} \\ k_1 \\ k_2 \\ k_3 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \begin{bmatrix} f_{1,1} & f_{1,2} \\ f_{2,1} & f_{2,2} \\ f_{3,1} & f_{3,2} \end{bmatrix} \end{array}$$

where each $f_{i,j}$ element stands for the frequency of term k_i in document d_j

Basic Concepts

- Logical view of a document: from full text to a set of index terms



The Boolean Model

The Boolean Model

- Simple model based on **set theory** and **boolean algebra**
- Queries specified as boolean expressions
  - quite intuitive and precise semantics
  - neat formalism
  - example of query

$$q = k_a \wedge (k_b \vee \neg k_c)$$

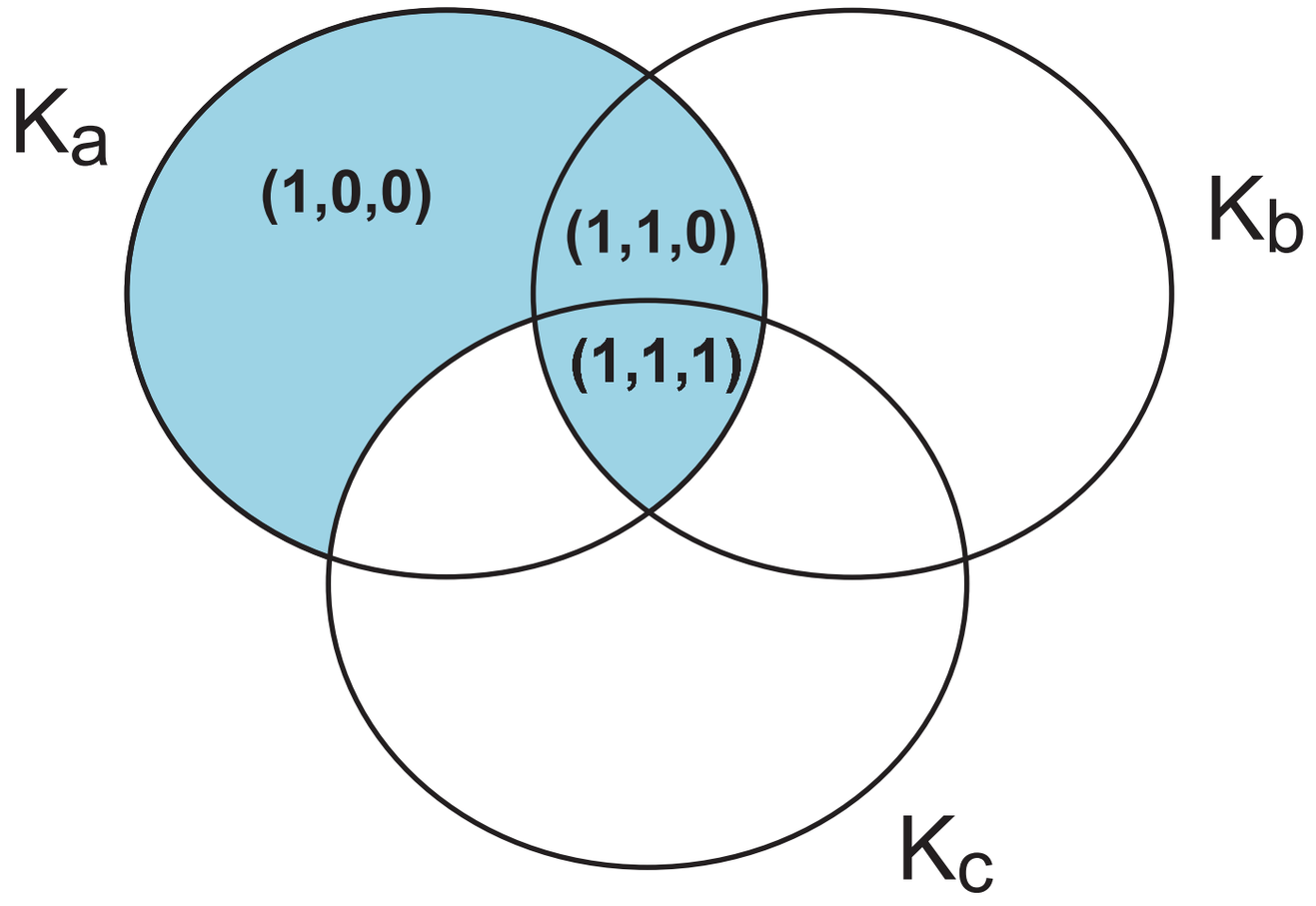
- Term-document frequencies in the term-document matrix are all binary
  - $w_{ij} \in \{0, 1\}$: weight associated with pair (k_i, d_j)
  - $w_{iq} \in \{0, 1\}$: weight associated with pair (k_i, q)

The Boolean Model

- A term conjunctive component that satisfies a query q is called a **query conjunctive component** $c(q)$
- A query q rewritten as a disjunction of those components is called the **disjunct normal form** q_{DNF}
- To illustrate, consider
  - query $q = k_a \wedge (k_b \vee \neg k_c)$
  - vocabulary $V = \{k_a, k_b, k_c\}$
- Then
  - $q_{DNF} = (1,1,1) \vee (1,1,0) \vee (1,0,0)$
  - $c(q)$: a conjunctive component for q

The Boolean Model

- The three conjunctive components for the query $q = k_a \wedge (k_b \vee \neg k_c)$



The Boolean Model

- This approach works even if the vocabulary of the collection includes terms not in the query
- Consider that the vocabulary is given by $V = \{k_a, k_b, k_c, k_d\}$
- Then, a document d_j that contains only terms k_a, k_b, and k_c is represented by $c(d_j) = (1, 1, 1, 0)$
- The query $[q = k_a \ \wedge \ (k_b \ \vee \ \neg k_c)]$ is represented in disjunctive normal form as

$$\begin{aligned} q_{DNF} \ = \ & (1,1,1,0) \vee (1,1,1,1) \vee \\ & (1,1,0,0) \vee (1,1,0,1) \vee \\ & (1,0,0,0) \vee (1,0,0,1) \end{aligned}$$

The Boolean Model

- The similarity of the document d_j to the query q is defined as

$$sim(d_j, q) = \begin{cases} 1 & \text{if } \exists c(q) \mid c(q) = c(d_j) \\ 0 & \text{otherwise} \end{cases}$$

- The Boolean model predicts that each document is either relevant or non-relevant

Drawbacks of the Boolean Model

- Retrieval based on binary decision criteria with no notion of partial matching
- No ranking of the documents is provided (absence of a grading scale)
- Information need has to be translated into a Boolean expression, which most users find awkward
- The Boolean queries formulated by the users are most often too simplistic
- The model frequently returns either too few or too many documents in response to a user query

Term Weighting

Term Weighting

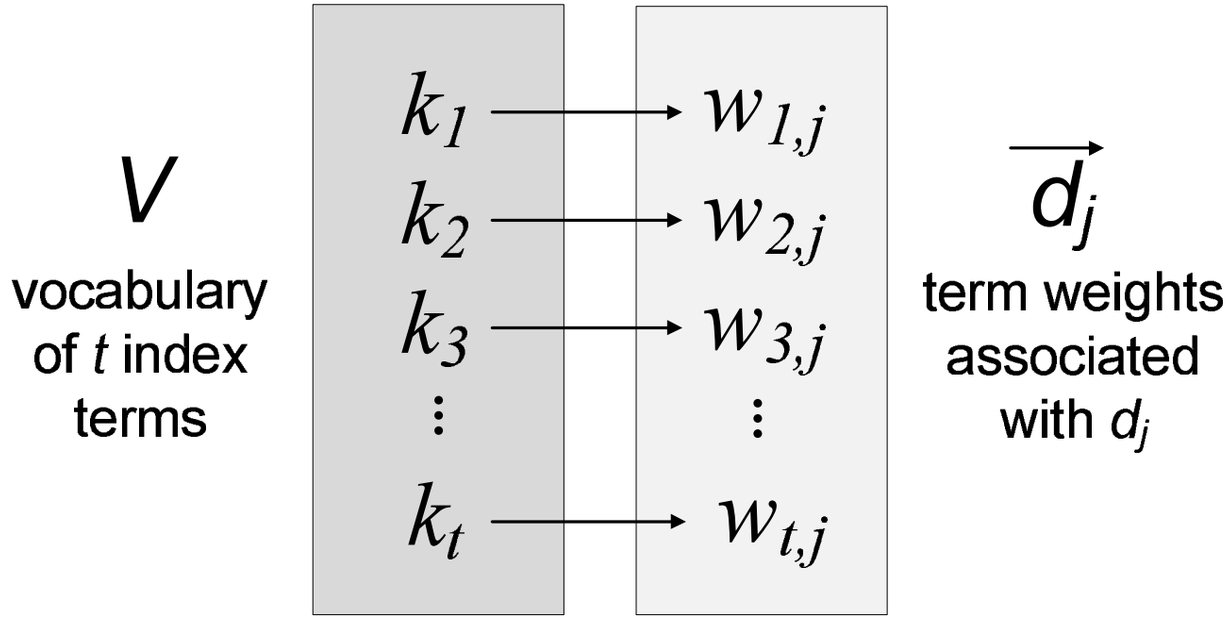
- The terms of a document are not equally useful for describing the document contents
- In fact, there are index terms which are simply vaguer than others
- There are properties of an index term which are useful for evaluating the importance of the term in a document
- For instance, a word which appears in all documents of a collection is completely useless for retrieval tasks

Term Weighting

- To characterize term importance, we associate a weight $w_{i,j} > 0$ with each term k_i that occurs in the document d_j
  - If k_i that does not appear in the document d_j, then $w_{i,j} = 0$.
- The weight $w_{i,j}$ quantifies the importance of the index term k_i for describing the contents of document d_j
- These weights are useful to compute a rank for each document in the collection with regard to a given query

Term Weighting

- Let,
  - k_i be an index term and d_j be a document
  - $V = \{k_1, k_2, ..., k_t\}$ be the set of all index terms
  - $w_{i,j} \geqslant 0$ be the weight associated with (k_i, d_j)
- Then we define $\vec{d_j} = (w_{1,j}, w_{2,j}, ..., w_{t,j})$ as a weighted vector that contains the weight $w_{i,j}$ of each term $k_i \in V$ in the document d_j



Term Weighting

- The weights $w_{i,j}$ can be computed using the **frequencies of occurrence** of the terms within documents
- Let $f_{i,j}$ be the frequency of occurrence of index term k_i in the document d_j
- The **total frequency of occurrence** F_i of term k_i in the collection is defined as

$$F_i = \sum_{j=1}^{N} f_{i,j}$$

where N is the number of documents in the collection

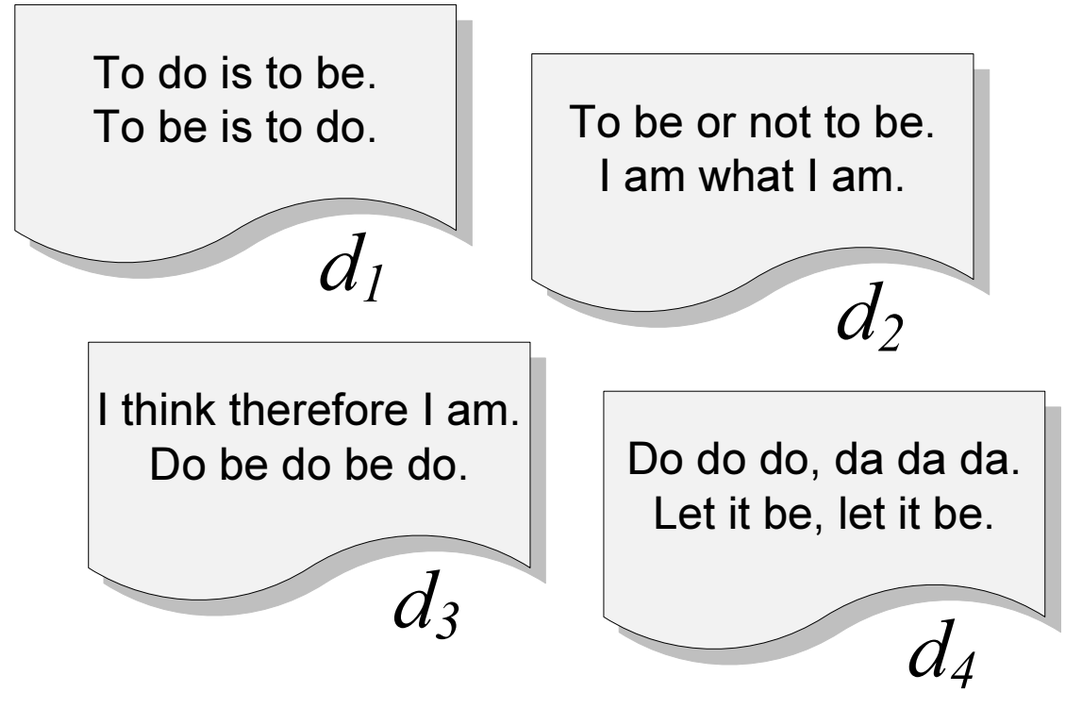
Term Weighting

- The **document frequency** n_i of a term k_i is the number of documents in which it occurs
  - Notice that $n_i \leq F_i$.
- For instance, in the document collection below, the values $f_{i,j}$, F_i and n_i associated with the term *do* are

$$f(do, d_1) = 2$$
$$f(do, d_2) = 0$$
$$f(do, d_3) = 3$$
$$f(do, d_4) = 3$$

$$F(do) = 8$$

$$n(do) = 3$$



Why is document frequency important?

TF-IDF Weights

TF-IDF Weights

- TF-IDF term weighting scheme:
  - Term frequency (TF)
  - Inverse document frequency (IDF)
  - Foundations of the most popular term weighting scheme in IR

Term Frequency (TF) Weights

- A variant of tf weight used in the literature is

$$tf_{i,j} = \begin{cases} 1 + \log f_{i,j} & \texttt{if} \quad f_{i,j} > 0 \\ 0 & \texttt{otherwise} \end{cases}$$

where the log is taken in base 2

- The log expression is a the preferred form because it makes them directly comparable to idf weights, as we later discuss

Term Frequency (TF) Weights

- Log tf weights $tf_{i,j}$ for the example collection

To do is to be.
To be is to do.

d_1

To be or not to be.
I am what I am.

d_2

I think therefore I am.
Do be do be do.

d_3

Do do do, da da da.
Let it be, let it be.

d_4

Vocabulary		$tf_{i,1}$	$tf_{i,2}$	$tf_{i,3}$	$tf_{i,4}$
1	to	3	2	-	-
2	do	2	-	2.585	2.585
3	is	2	-	-	-
4	be	2	2	2	2
5	or	-	1	-	-
6	not	-	1	-	-
7	I	-	2	2	-
8	am	-	2	1	-
9	what	-	1	-	-
10	think	-	-	1	-
11	therefore	-	-	1	-
12	da	-	-	-	2.585
13	let	-	-	-	2
14	it	-	-	-	2

Inverse Document Frequency

- We call **document exhaustivity** the number of index terms assigned to a document
- The more index terms are assigned to a document, the higher is the probability of retrieval for that document
  - If too many terms are assigned to a document, it will be retrieved by queries for which it is not relevant
- **Optimal exhaustivity**. We can circumvent this problem by optimizing the number of terms per document
- Another approach is by weighting the terms differently, by exploring the notion of **term specificity**

Inverse Document Frequency

- **Specificity** is a property of the term semantics
  - A term is more or less specific depending on its meaning
  - To exemplify, the term *beverage* is less specific than the terms *tea* and *beer*
  - We could expect that the term *beverage* occurs in more documents than the terms *tea* and *beer*
- Term specificity should be interpreted as a statistical rather than semantic property of the term
- **Statistical term specificity**. The inverse of the number of documents in which the term occurs

Inverse Document Frequency

- Let k_i be the term with the rth largest document frequency, i.e., $n(r) = n_i$. Then,

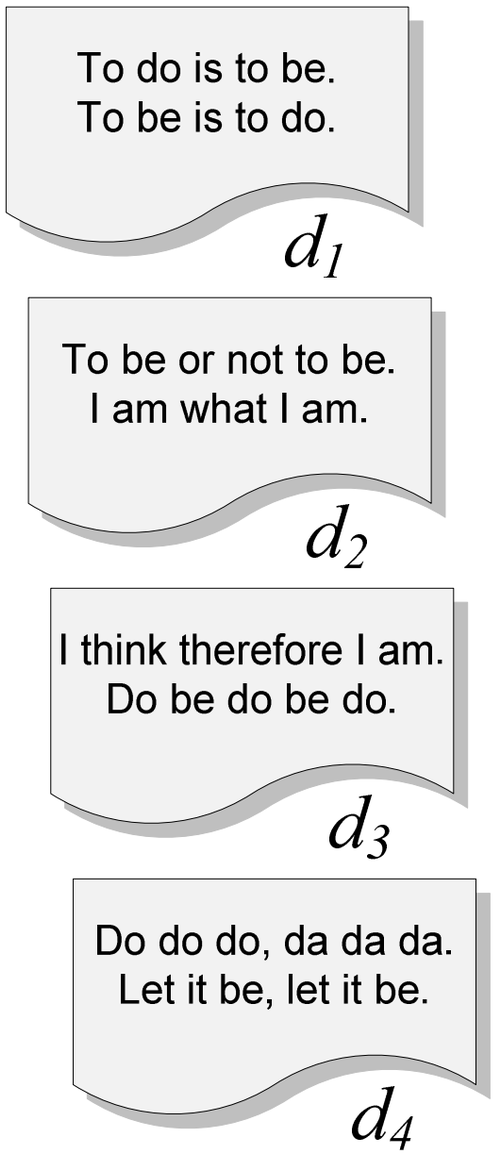
$$idf_i = \log \frac{N}{n_i}$$

where idf_i is called the **inverse document frequency** of term k_i

- Idf provides a foundation for modern term weighting schemes and is used for ranking in almost all IR systems

Inverse Document Frequency

- Idf values for example collection



	term	n_i	$idf_i = \log(N/n_i)$
1	to	2	1
2	do	3	0.415
3	is	1	2
4	be	4	0
5	or	1	2
6	not	1	2
7	I	2	1
8	am	2	1
9	what	1	2
10	think	1	2
11	therefore	1	2
12	da	1	2
13	let	1	2
14	it	1	2

TF-IDF weighting scheme

- The best known term weighting schemes use weights that combine idf factors with term frequencies
- Let $w_{i,j}$ be the term weight associated with the term k_i and the document d_j
- Then, we define

$$w_{i,j} = \begin{cases} (1 + \log f_{i,j}) \times \log \frac{N}{n_i} & \texttt{if} \quad f_{i,j} > 0 \\ 0 & \texttt{otherwise} \end{cases}$$

which is referred to as a **tf-idf weighting scheme**

TF-IDF weighting scheme

- Tf-idf weights of all terms present in our example document collection

To do is to be.
To be is to do.

d_1

To be or not to be.
I am what I am.

d_2

I think therefore I am.
Do be do be do.

d_3

Do do do, da da da.
Let it be, let it be.

d_4

		d_1	d_2	d_3	d_4
1	to	3	2	-	-
2	do	0.830	-	1.073	1.073
3	is	4	-	-	-
4	be	-	-	-	-
5	or	-	2	-	-
6	not	-	2	-	-
7	I	-	2	2	-
8	am	-	2	1	-
9	what	-	2	-	-
10	think	-	-	2	-
11	therefore	-	-	2	-
12	da	-	-	-	5.170
13	let	-	-	-	4
14	it	-	-	-	4

Document Length Normalization

Document Length Normalization

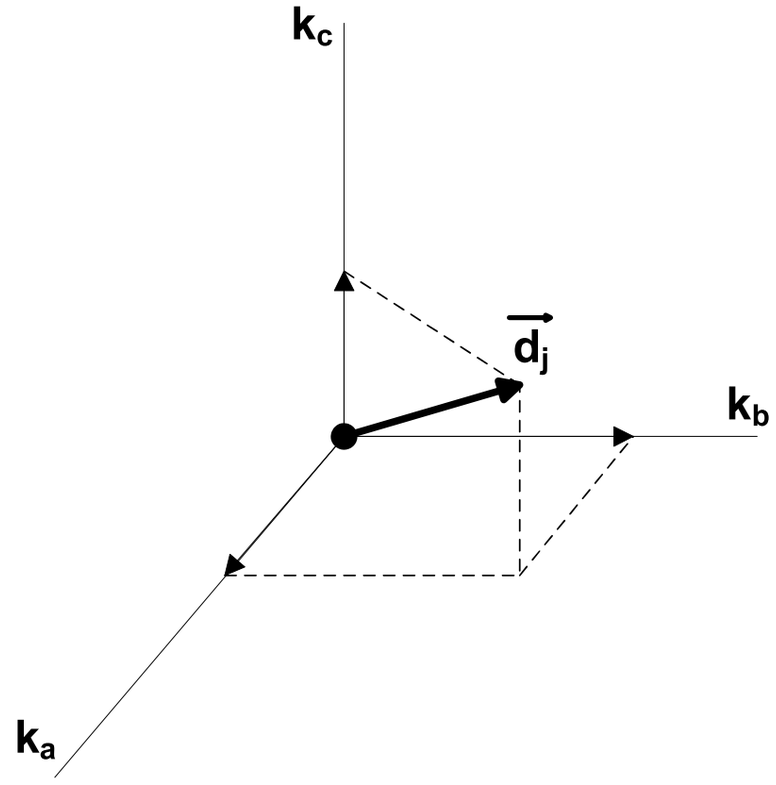
- Document sizes might vary widely
- This is a problem because longer documents are more likely to be retrieved by a given query
- To compensate for this undesired effect, we can divide the rank of each document by its length
- This procedure consistently leads to better ranking, and it is called **document length normalization**

Document Length Normalization

- Methods of document length normalization depend on the representation adopted for the documents:
  - **Size in bytes**: consider that each document is represented simply as a stream of bytes
  - **Number of words**: each document is represented as a single string, and the document length is the number of words in it
  - **Vector norms**: documents are represented as vectors of weighted terms

Document Length Normalization

- Documents represented as vectors of weighted terms
  - Each term of a collection is associated with an orthonormal unit vector $\vec{k}_i$ in a t-dimensional space
  - For each term k_i of a document d_j is associated the term vector component $w_{i,j} \times \vec{k}_i$



Document Length Normalization

- The document representation $\vec{d}_j$ is a vector composed of all its term vector components

$$\vec{d}_j = (w_{1,j}, w_{2,j}, ..., w_{t,j})$$

- The document length is given by the norm of this vector, which is computed as follows

$$|\vec{d}_j| = \sqrt{\sum_{i}^{t} w_{i,j}^2}$$

Document Length Normalization

- Three variants of document lengths for the example collection

To do is to be.
To be is to do.

d_1

To be or not to be.
I am what I am.

d_2

I think therefore I am.
Do be do be do.

d_3

Do do do, da da da.
Let it be, let it be.

d_4

	d_1	d_2	d_3	d_4
size in bytes	34	37	41	43
number of words	10	11	10	12
vector norm	5.068	4.899	3.762	7.738

The Vector Model

The Vector Model

- Boolean matching and binary weights is too limiting
- The vector model proposes a framework in which partial matching is possible
- This is accomplished by assigning non-binary weights to index terms in queries and in documents
- Term weights are used to compute a **degree of similarity** between a query and each document
- The documents are **ranked** in decreasing order of their degree of similarity

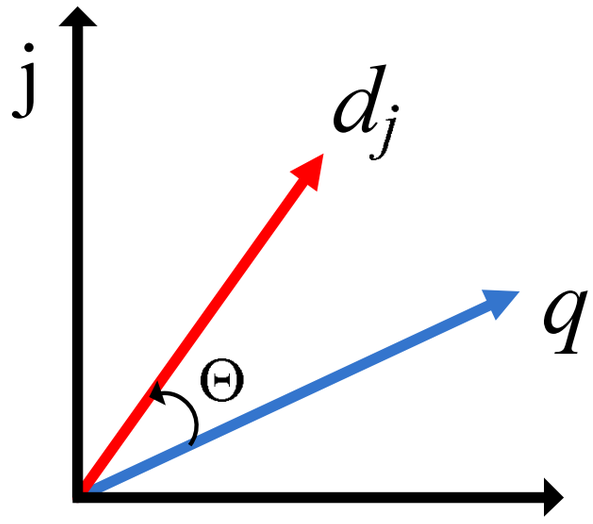
The Vector Model

- For the vector model:
  - The weight $w_{i,j}$ associated with a pair (k_i, d_j) is positive and non-binary
  - The index terms are assumed to be all mutually independent
  - They are represented as unit vectors of a t-dimensionsal space (t is the total number of index terms)
  - The representations of document d_j and query q are t-dimensional vectors given by

$$\vec{d_j} = (w_{1j}, w_{2j}, \ldots, w_{tj})$$
$$\vec{q} = (w_{1q}, w_{2q}, \ldots, w_{tq})$$

The Vector Model

- Similarity between a document d_j and a query q



i $\quad cos(\theta) = \frac{d_j \bullet \vec{q}}{|\vec{d_j}| \times |\vec{q}|}$

$$sim(d_j, q) = \frac{\sum_{i=1}^{t} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^2} \times \sqrt{\sum_{j=1}^{t} w_{i,q}^2}}$$

Since $w_{ij} > 0$ and $w_{iq} > 0$, we have $0 \leqslant sim(d_j, q) \leqslant 1$

The Vector Model

- Weights in the Vector model are basically tf-idf weights

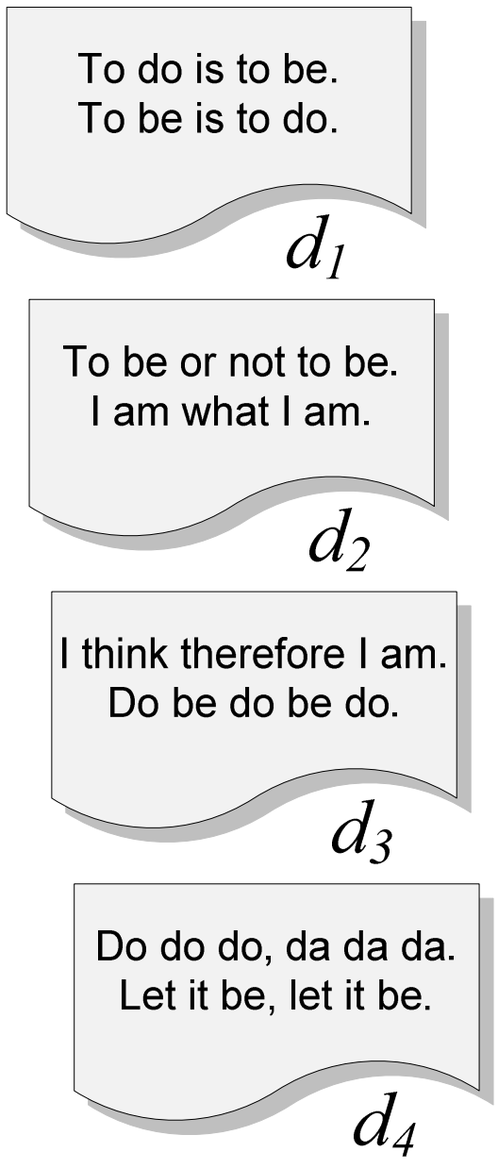
$$w_{i,q} = (1 + \log f_{i,q}) \times \log \frac{N}{n_i}$$

$$w_{i,j} = (1 + \log f_{i,j}) \times \log \frac{N}{n_i}$$

- These equations should only be applied for values of term frequency greater than zero
- If the term frequency is zero, the respective weight is also zero

The Vector Model

- Document ranks computed by the Vector model for the query “to do” (see tf-idf weight values in Slide 43)



doc	rank computation	rank
d_1	$\frac{1*3+0.415*0.830}{5.068}$	0.660
d_2	$\frac{1*2+0.415*0}{4.899}$	0.408
d_3	$\frac{1*0+0.415*1.073}{3.762}$	0.118
d_4	$\frac{1*0+0.415*1.073}{7.738}$	0.058

Note!

- We assume that the query does not have any effect (or will have a very small effect) on the IDF

- Therefore, for each query term, we use the same IDF used for the documents

The Vector Model

- Advantages:
  - term-weighting improves quality of the answer set
  - partial matching allows retrieval of docs that approximate the query conditions
  - cosine ranking formula sorts documents according to a degree of similarity to the query
  - document length normalization is naturally built-in into the ranking
- Disadvantages:
  - It assumes independence of index terms

Vector Space Models in Python

Note

- In principle, you can use what you have learned so far to implement vector space models in Python

- However, we will now also see how to leverage existing functions available in libraries such as nltk and scikit-learn

Simple example

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics.pairwise import cosine_similarity

#initialize the vectorizer
vectorizer=CountVectorizer()

#Add the files in the corpus, a document on each line
corpus=["Racing games",
        "This document describes racing cars",
        "This document is about video games in general",
        "This is a nice racing video game"]

#creates the model
model=vectorizer.fit_transform(corpus)
```

CountVectorizer

- Class that automatically builds vector space models from a corpus (list) of documents

- Features available for stop word removal, stemming, customized tokenization

- Documentation here: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

Model converted as an array, and feature list…

```
print(model.shape)
print(vectorizer.get_feature_names_out())
print(model.toarray())
```

```
(4, 13)
[‘about' 'cars' 'describes' 'document' 'game'
'games' 'general' 'in' 'is'
 'nice' 'racing' 'this' 'video']
[[0 0 0 0 0 1 0 0 0 0 1 0 0]
 [0 1 1 1 0 0 0 0 0 0 1 1 0]
 [1 0 0 1 0 1 1 1 1 0 0 1 1]
 [0 0 0 0 1 0 0 0 1 1 1 1 1]]
```

Discussion

- We print
    - Model shape (number of documents and vocabulary words)
    - The vocabulary
    - The model as a matrix of frequencies
- We can note that stop word removal and stemming were not performed…

Better version…

- Including stop word removal from NLTK

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from nltk.corpus import stopwords

#stop words
sw=stopwords.words('english')

#initialize the vectorizer
vectorizer=CountVectorizer(stop_words=sw)
```

Custom Tokenizer

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import snowball
import re

def my_tokenizer(text):
        sw=stopwords.words('english')
        stemmer=snowball.SnowballStemmer(language="english")
        tokens=word_tokenize(text)
        pruned=[stemmer.stem(t) for t in tokens if re.search(r"^\w",t) and not t in sw]
        return pruned
```

Custom tokenizer (cont.)

```
#initialize the vectorizer
vectorizer=CountVectorizer(tokenizer=my_tokenizer)

#Add the files in the corpus, a document on each line
corpus=["Racing games",
        "This document describes racing cars",
        "This document is about video games in general",
        "This is a nice racing video game"]

#creates the model
model=vectorizer.fit_transform(corpus)

#prints the model
print(model.shape)
print(vectorizer.get_feature_names_out())
print(model.toarray())
```

Computing cosine similarity

```
cos=cosine_similarity(model)
print(cos)
```

```
[[1.         0.35355339 0.35355339 0.70710678]
 [0.35355339 1.         0.25       0.25      ]
 [0.35355339 0.25       1.         0.5       ]
 [0.70710678 0.25       0.5        1.        ]]
```

Discussion

The most similar document to the query (first document) is the last one

Tf-Idf model

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import snowball
import re

def my_tokenizer(text):
        sw=stopwords.words('english')
        stemmer=snowball.SnowballStemmer(language="english")
        tokens=word_tokenize(text)
        pruned=[stemmer.stem(t) for t in tokens if re.search(r"^\w",t) and not t in sw]
        return pruned

#initialize the vectorizer
vectorizer=TfidfVectorizer(tokenizer=my_tokenizer)

#Add the files in the corpus, a document on each line
corpus=["Racing games",
        "This document describes racing cars",
        "This document is about video games in general",
        "This is a nice racing video game"]

#creates the model
model=vectorizer.fit_transform(corpus)
```

Tf-Idf Model

```
print(model.shape)
print(vectorizer.get_feature_names_out())
print(model.toarray())
```

```
(4, 8)
['car' 'describ' 'document' 'game' 'general' 'nice' 'race' 'video']
[[0.         0.         0.         0.70710678 0.         0.  0.70710678 0.        ]
 [0.57457953 0.57457953 0.4530051  0.         0.         0.  0.36674667 0.        ]
 [0.         0.         0.4842629  0.39205255 0.61422608 0.  0.         0.4842629 ]
 [0.         0.         0.         0.40892206 0.         0.64065543  0.40892206 0.5051001 ]]
```

Cosine similarity

```
cos=cosine_similarity(model)
print(cos)
```

```
 [[1.         0.25932906 0.27722302 0.57830313]
  [0.25932906 1.         0.21937356 0.1499708 ]
  [0.27722302 0.21937356 1.         0.40492018]
  [0.57830313 0.1499708  0.40492018 1.        ]]
```

Discussion

- With this implementation, we can simply convert a set of documents into a vector space model

- And then, compute pairwise similarity

- What if we have already a set of documents and we need to compare a query with the documents?

Example with external query

- You can use the transform function

- Note: it takes a string, so you need to preprocess its words and recompose it

```
#adds a query to the model
query=vectorizer.transform(["racing game"])
print(query.toarray())


cos=cosine_similarity(query,model)
print(cos)
```

Results…

- Vector content:

```
[[0.         0.         0.         0.70710678 0.         0.  0.70710678 0.        ]]
```

- Similarity with respect to

```
corpus=["This document describes racing cars",
        "This document is about video games in general",
        "This is a nice racing video game"]
```

```
[[0.30267425 0.32516555 0.6503311 ]]
```

Vector Space Implementation Issues

Sparse Vectors

- Vocabulary and therefore dimensionality of vectors can be very large, ~10^4 .
- However, most documents and queries do not contain most words, so vectors are sparse (i.e., most entries are 0).
- Need efficient methods for storing and computing with sparse vectors.

Sparse Vectors as Lists

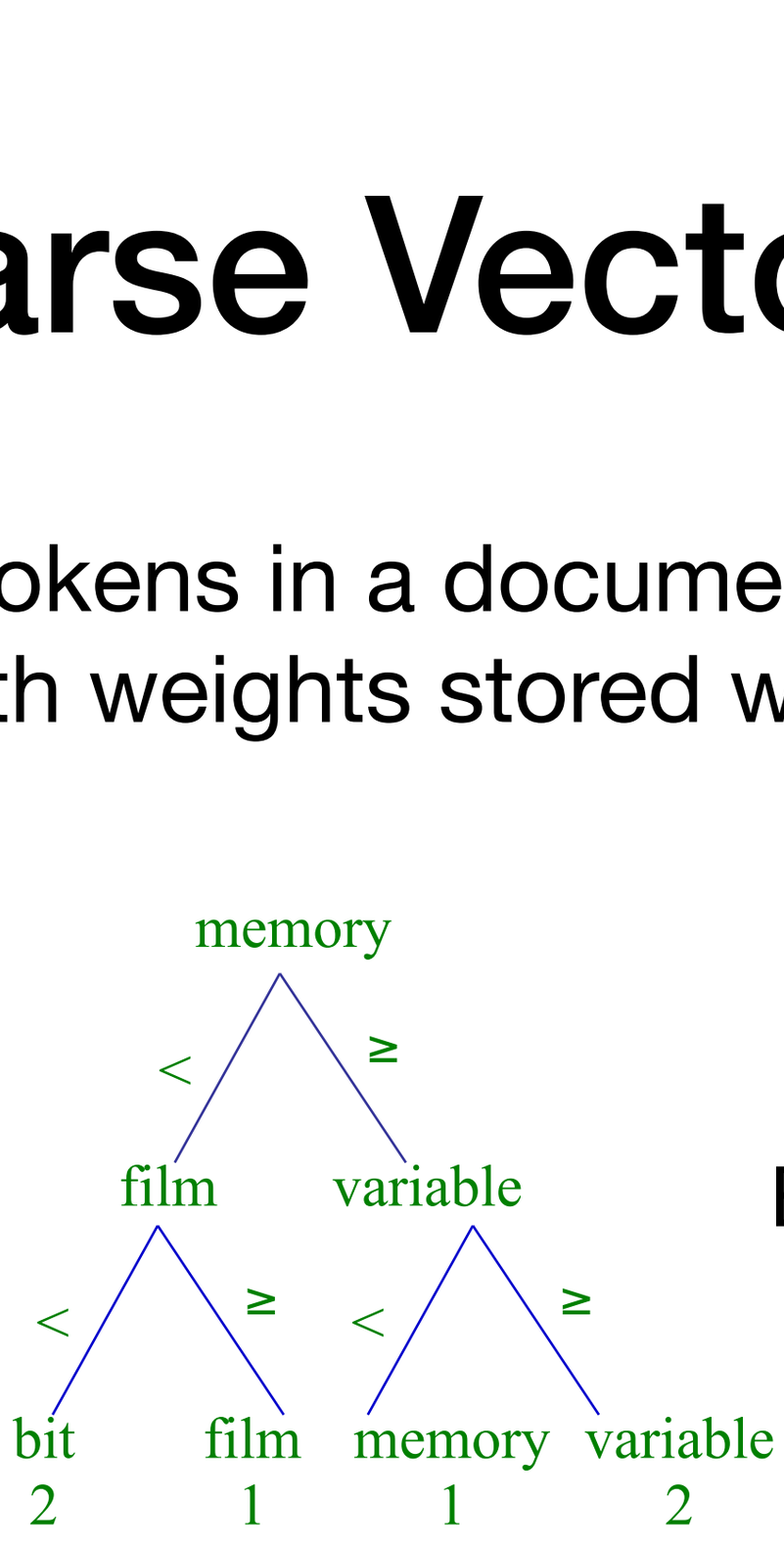
Store vectors as linked lists of non-zero-weight tokens paired with a weight.

- Space proportional to number of unique tokens (n) in document.
- Requires linear search of the list to find (or change) the weight of a specific token.
- Requires quadratic time in worst case to compute vector for a document:

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

Sparse Vectors as Trees

Index tokens in a document in a balanced binary tree with weights stored with tokens at the leaves.



Balanced Binary Tree

Sparse Vectors as Trees (cont.)

- Space overhead for tree structure: ~2n nodes.
- O(log n) time to find or update weight of a specific token.
- O(n log n) time to construct vector.
- Need software package to support such data structures.

Sparse Vectors as HashTables

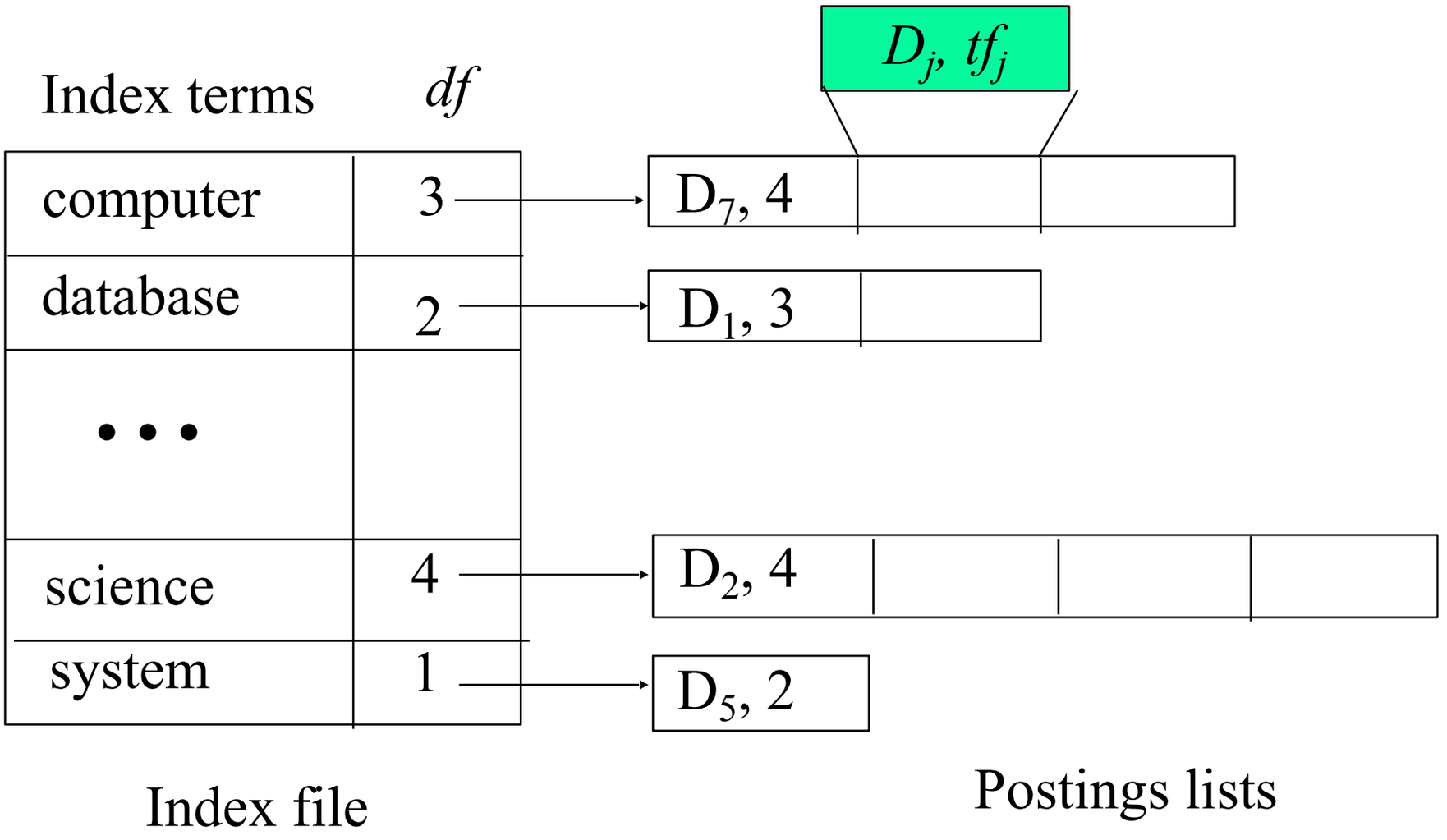
Store tokens in hashtable, with token string as key and weight as value.

- Storage overhead for hashtable ~1.5n.
- Table must fit in main memory.
- Constant time to find or update weight of a specific token (ignoring collisions).
- O(n) time to construct vector (ignoring collisions).

Implementation Based on Inverted Files

- In practice, document vectors are not stored directly
  - an inverted organization provides much better efficiency.
- The keyword-to-document index can be implemented as a hash table, a sorted array, or a tree-based data structure (trie, B-tree).
- Critical issue is logarithmic or constant-time access to token information

Inverted Index



Query: “computer science”

- Computer: IDF= log(100/3)
  - D7: 3
  - D5: 5
  - D2: 1
- Science: IDF = log (100/4)
  - D2: 3
  - D4: 2
  - D5: 3
  - D7: 8

- D2: log(100/3) *1*1* log(100/3) + log (100/4)*1+3*log (100/4)
- D4:
- D5:log(100/3) *1*5* log(100/3)
- D7: log(100/3) *1*3* log(100/3)

sqrt(1* log(100/3)*1* log(100/3) +1* log (100/4)*1* log (100/4))

Retrieval with an Inverted Index

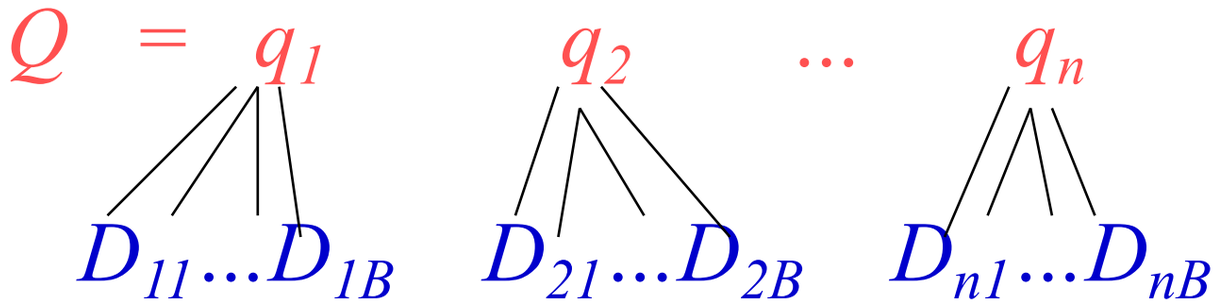
- Tokens that are not in both the query and the document do not effect cosine similarity.
  - Product of token weights is zero and does not contribute to the dot product.

- Usually the query is fairly short, and therefore its vector is extremely sparse.

- Use inverted index to find the limited set of documents that contain at least one of the query words.

Inverted Query Retrieval Efficiency

- Assume that, on average, a query word appears in B documents:



- Then retrieval time is O(|Q| B), which is typically, *much* better than naïve retrieval that examines all N documents, O(|V| N), because |Q| << |V| and B << N.

Processing the Query

- Incrementally compute cosine similarity of each indexed document as query words are processed one by one.
- To accumulate a total score for each retrieved document, store retrieved documents in a hashtable, where DocumentReference is the key and the partial accumulated score is the value.

Inverted-Index Retrieval Algorithm

```
Create a HashMapVector, Q, for the query.
Create empty HashMap, R, to store retrieved documents with scores.
For each token, T, in Q:
     Let I be the IDF of T, and K be the count of T in Q
     Set the weight of T in Q:   W = K * I
     Let L be the list of TokenOccurences of T from H
     For each TokenOccurence, O, in L:
         Let D be the document of O, and C be the count of O (tf of T in D)
        if D is not already in R (D was not previously retrieved)
                Then add D to R and initialize score to 0.0
         Increment D's score by W * I * C (product of T-weight in Q and D)
```

Retrieval Algorithm (cont)

```
Compute the length, L, of the vector Q (square-root of the sum of the
   squares of its weights)

For each retrieved document D in R:
      Let S be the current accumulated score of D
        (S is the dot-product of D and Q)
      Let Y be the length of D as stored in its DocumentReference
      Normalize D's final score to S/(L * Y)

Sort retrieved documents in R by final score and return results in an
   array.
```

Efficiency Note

To save computation and an extra iteration through the tokens in the query, the computation of the length of the query vector is integrated with the processing of query tokens during retrieval.

Implementation in scikit-learn

- You can use the HashingVectorizer class
- It can be used similarly to the CountVectorizer
- Cons: it doesn’t use tf-idf as it would make the representation stateful

Alternative Implementation

- Available in the gensim package

- Another very useful package for NLP in Python

Gensim example

```
from gensim import corpora
from gensim import models
from gensim import similarities
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import snowball
import re

def my_tokenizer(text):
    """tokenization function"""
    sw=stopwords.words('english')
    stemmer=snowball.SnowballStemmer(language="english")
    tokens=word_tokenize(text)
    pruned=[stemmer.stem(t.lower()) for t in tokens \
            if re.search(r"^\w",t) and not t.lower() in sw]
    return pruned

documents=["This document describes racing cars",
       "This document is about video games in general",
       "This is a nice racing video game"]
```

Gensim Example (cont.)

```
texts=[my_tokenizer(d) for d in documents]

#creates the dictionary for the document corpus
dictionary = corpora.Dictionary(texts)
#creates a bag of word corpus
bow_corpus=[dictionary.doc2bow(text) for text in texts]

#creates a tf-idf model from the bag of word corpus
tfidf = models.TfidfModel(bow_corpus)
#creates an index that facilitates the computation of similarities
index = similarities.SparseMatrixSimilarity(tfidf[bow_corpus],len(dictionary))
print(list(index))
```

Running the query…

```
#tokenizes the query
query_document = my_tokenizer("racing games")
#indexes the query using the documents' dictionary
query_bow = dictionary.doc2bow(query_document)

#computes the similarity between the query and the documents
sims = index[tfidf[query_bow]]
print(list(enumerate(sims)))
```

Results…

```
[(0, 0.17312077), (1, 0.21988432), (2, 0.43976864)]
```