course:

Searching the Web (NDBI038)

Searching the Web and Multimedia Databases (BI-VWM)

lecture 3:

Vector model of information retrieval

prof. RNDr. Tomáš Skopal, Ph.D.
Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague
Department of Software Engineering, Faculty of Information Technology, Czech Technical University in Prague

Today's lecture outline

- vector model
  - motivation
  - term weighting
  - querying
  - implementation by inverted index
- latent semantic indexing
  - motivation
  - singular-value decomposition
  - Querying
- word2vec

Vector model – motivation

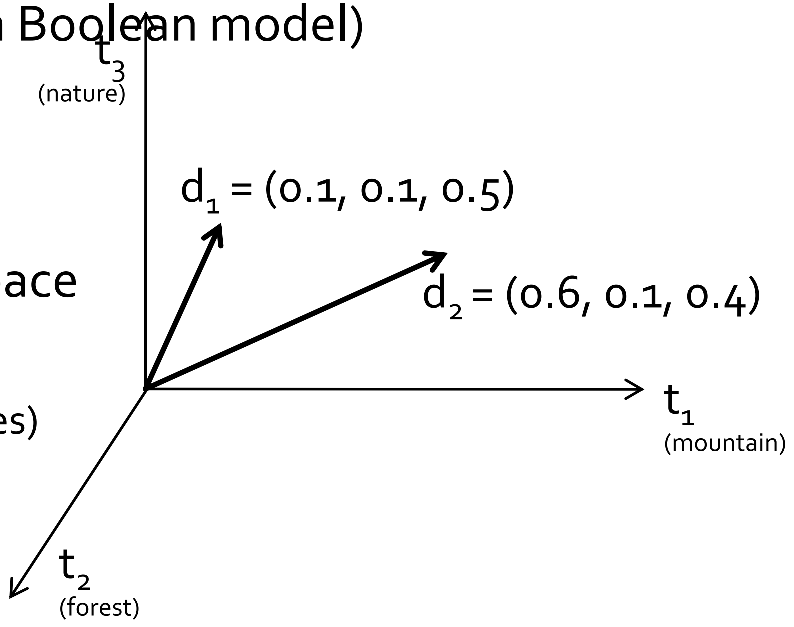
- content-based similarity search model
  - query is represented the same as document
    – by a vector in a high-dimensional space
  - the **distance** of query vector to a document vector
    ≈ **dissimilarity** of the query text and document text
    ≈ the **relevance** of the document to the query
  - cognitive model – people **search for similar** things

Vector model – motivation

- highly parameterizable
  - different term representation schemas (positioning the vectors in space)
  - different similarity (distance) measures
- ranking of documents in the query result
- support of relevance feedback

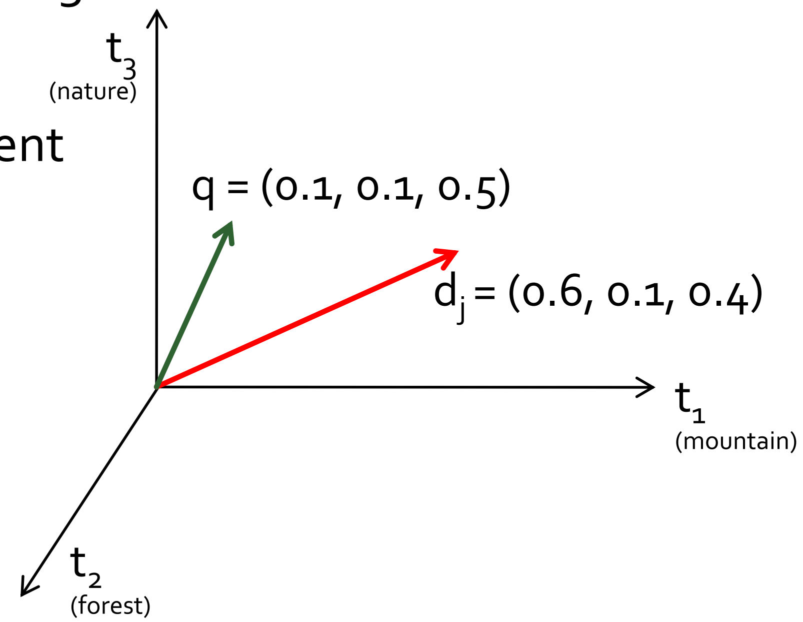
Vector model – the basics

- also known as the **bag of words** (BoW) model
- document = bag of terms
  - bag = multi-set = set allowing multiple occurrences of the same element
  - vocabulary of **m** terms (the same as in Boolean model)
  - document $\mathbf{d}_j$ modeled by a vector of dimension **m**
  - extremely high-dimensional vector space
    - note that usually **m** > 10^4 for an English collection (could be more for other languages)



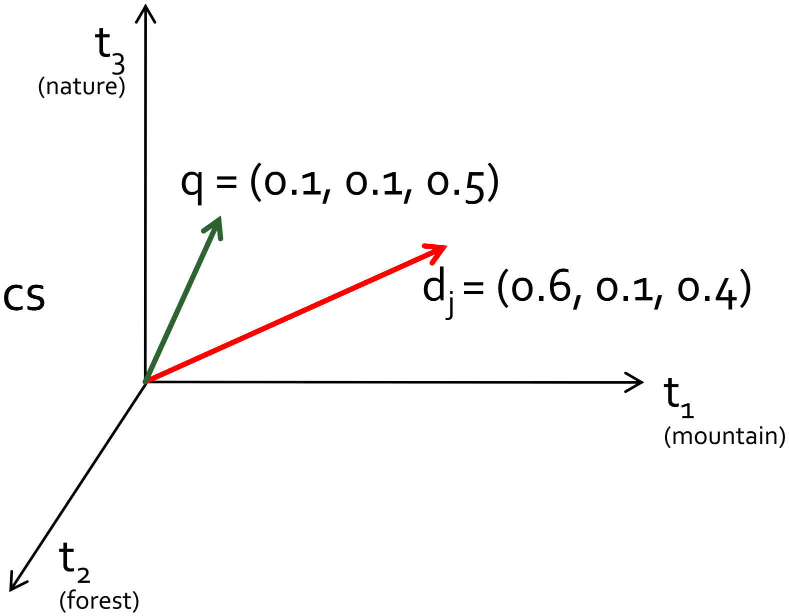
Vector model – the basics

- query = bag of terms, i.e., the same as document
  - could be specified by **a few keywords** (e.g., typing into Google) or by a **query document** (e.g., searching for plagiarism)
  - no Boolean condition or a query language
  - query modeled by a vector **q** of dimension **m**, the same as document



Vector model – the basics

- the space structure
  - each **dimension** of the space belongs to a **term** from the vocabulary
  - a coordinate value **x** in a dimension **i** means the weight **x** (importance) of term $\mathbf{t}_i$
- document/query vector
  - point (vector) in the space represents a unique combination of term weights
  - describes the basic term-based statistics of a document/query
    - a compact descriptor
    - quite well defined semantics



Vector model – the basics

- term-by-document matrix A
  - similar to the one shown in last lecture (Boolean model), i.e., stores document vectors $\mathbf{d}_j$/term vectors $\mathbf{t}_i$
  - not binary values – occurrence of a term in documents, but real values – weight $\mathbf{w}_{ij}$ of a term $\mathbf{t}_i$ in document $\mathbf{d}_j$
- zero weight = a term has no significance in the document or it doesn't occur in the document
  - usually sparse matrix (99%)

$$\mathbf{A} = \begin{array}{c c} & \begin{array}{cccc} t_1 & t_2 & \cdots & t_m \end{array} \\ \begin{array}{c} d_1 \\ d_2 \\ \vdots \\ \vdots \\ d_n \end{array} & \left(\begin{array}{cccc} w_{11} & w_{21} & \cdots & w_{m1} \\ w_{12} & w_{22} & \cdots & w_{m2} \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ w_{1n} & w_{2n} & \cdots & w_{mn} \end{array} \right) \end{array}$$

Vector model – the term weights

- how to construct the term weights?
  - manual weighting leads to inconsistency and labor
  - automatic weighting based on term frequency statistics (in each document and in the entire collection)

- what is good weighting model?
  - high weights for **important** terms, and vice versa
    - important term
      = representative, discriminative, semantically significant term
  - low weights for indiscriminative terms (e.g., stop words) and terms appearing in many documents

Vector model – the term weights

- document-scope statistics
  - more frequent terms in a document are more important

    f_{ij} = **frequency of term t_i in document d_j**

  - normalized term frequency of term t_i in document d_j

    $tf_{ij} = f_{ij} / max_i\{f_{ij}\}$, where max returns the highest frequency of term t_i over the entire collection

    - weights normalized to 0..1
    - alone not robust enough, a document concatenated with itself would obtain double weights

Vector model – the term weights

- collection-scope statistics
  - terms present in many documents are less important

    df_i = **document frequency of term t_i**

    = number of documents containing term t_i

idf_i = **inverse document frequency of term t_i,**

$= \log_2 (n / df_i)$, where n is the total number of documents

Vector model – the term weights

- *tf-idf*, popular *weighting scheme*

  $$w_{ij} = tf_{ij}\, idf_i = tf_{ij} \log_2 (n / df_i)$$

  - the *idf* component generalizes the effect of removing stop words
    - logarithm used to inhibit the effect of term frequency (*tf*)
    - stop words and other frequent terms would obtain very low *idf*, and thus the entire weight
- pros
  - experimentally, *tf-idf* proved as the best w.r.t. precision/recall
- cons
  - using *idf* requires static collection for efficient query implementation (by inverted index)

- many other ways of determining term weights proposed

Vector model – the term weights

Example:

Given a document **d** containing terms with given frequencies in **d** as

d = < mountain(3), forest(2), nature(1) >

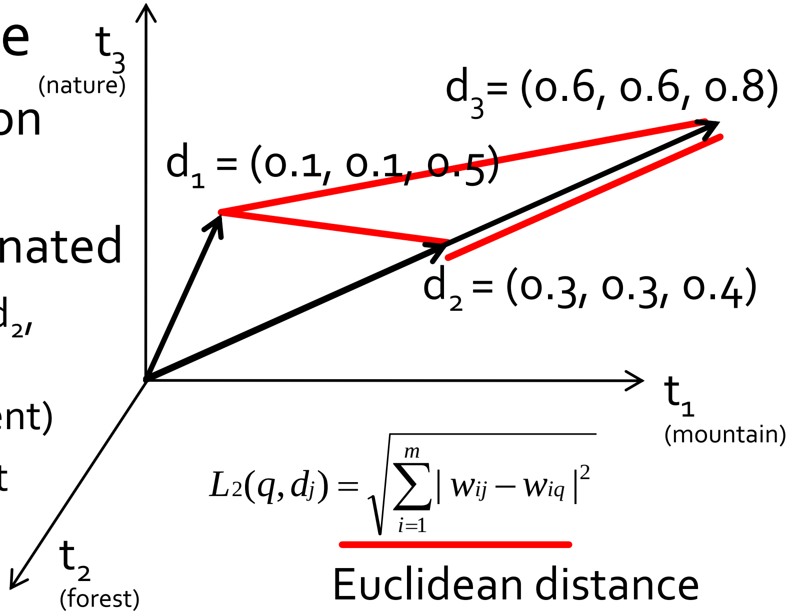
Assume 10,000 documents and document frequencies of the terms as mountain(50), forest(1300), nature(250)

Then the *tf-idf* weights of the terms in **d** are:

mountain:	$tf = 3/3$; $idf = \log_2(10000/50) = 7.6$;	**tf-idf = 7.6**
forest:	$tf = 2/3$; $idf = \log_2(10000/1300) = 2.9$;	**tf-idf = 2.0**
nature:	$tf = 1/3$; $idf = \log_2(10000/250) = 5.3$;	**tf-idf = 1.8**

Vector model – the similarity

- now we have the vector model and the weights, but what about the similarity of vectors?
- intuitivelly, one would choose the Euclidean (L_2) distance as the dissimilarity measure
- Euclidean distance not appropriate
  1. not robust w.r.t. weight multiplication e.g., consider documents d_2 and d_3, where d_3 is two copies of d_2 concatenated
     - although the text in d_3 is twice the same as d_2, the distance d_2 and d_3 is larger than d_2 and d_1 (where text in d_1 is completely different)
     - could be solved by vector normalization, but
  2. implementation problem (later)



$$L_2(q, d_j) = \sqrt{\sum_{i=1}^{m} | w_{ij} - w_{iq} |^2}$$

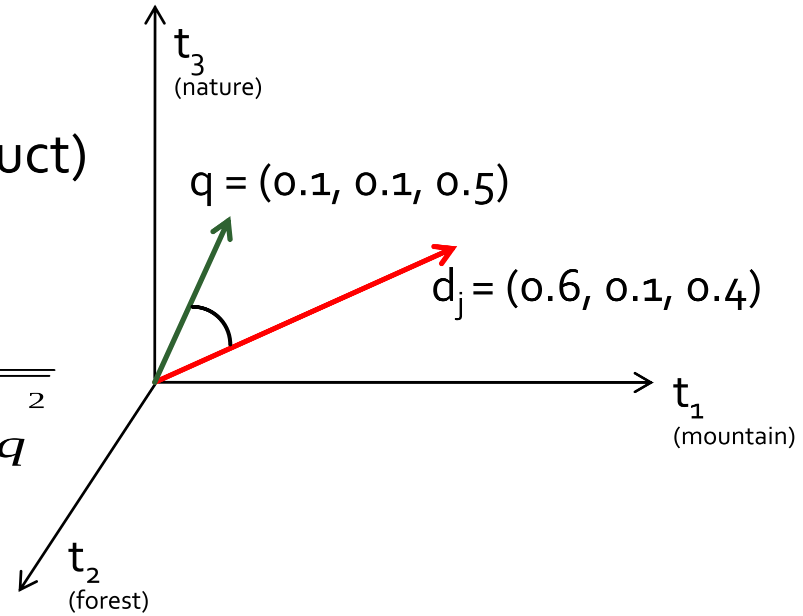
Euclidean distance between d_1, d_2, d_3

Vector model – the similarity

- robust similarity comparing the **directions (angles)**, not the **positions** (that is why we talk about vectors not points)
- inner product
  - sim($\boldsymbol{d_j}, \boldsymbol{q}$) = $\boldsymbol{d}_j \bullet \boldsymbol{q}$ = $\sum_{i=1}^{m} w_{ij} w_{iq}$
- cosine similarity (normed inner product)
  - CosSim($\boldsymbol{d_j}$, $\boldsymbol{q}$) =

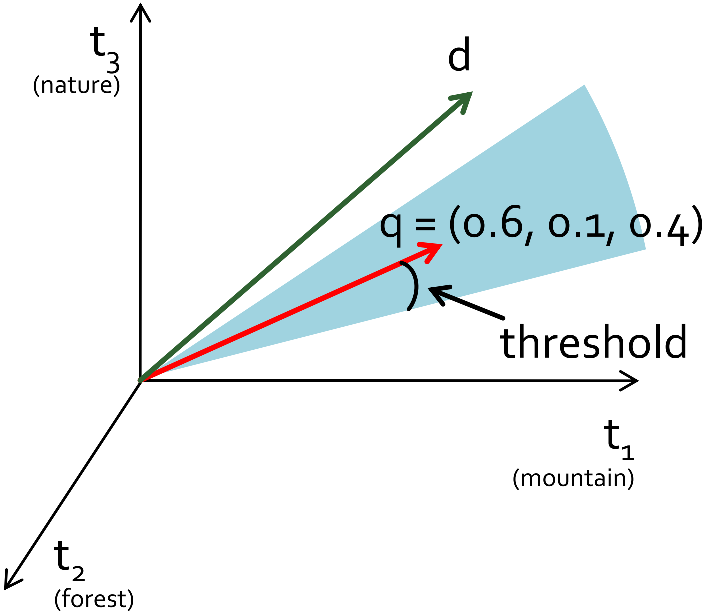
$$\frac{\vec{d}_j \cdot \vec{q}}{\left|\vec{d}_j\right| \cdot \left|\vec{q}\right|} = \frac{\sum_{i=1}^{m} (w_{ij} \cdot w_{iq})}{\sqrt{\sum_{i=1}^{m} w_{ij}^{2} \cdot \sum_{i=1}^{m} w_{iq}^{2}}}$$

  - cosine of the angle between **q** and $\mathbf{d}_j$ (high value = high similarity)
  - by arccos(CosSim) we obtain the **angle distance** (low dist.=high similarity)



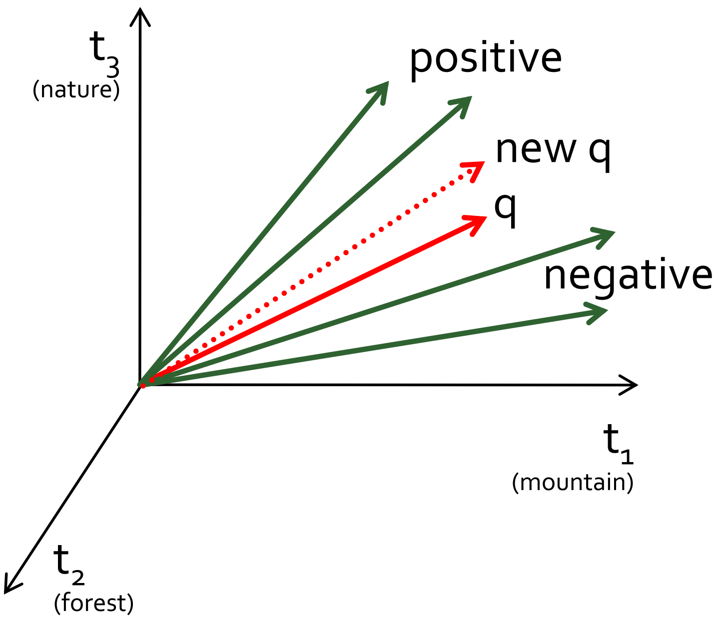
Vector model – querying

- **vector range query** used, specified by
  - a query vector **q**, that stands for the desired document
  - a **similarity threshold** that should be exceeded, when measuring the similarity of **q** and a document vector **d**
    - if CosSim(q,d) > threshold, then **d** goes to the query result
- the result of range query is a ranked set of documents, ordered by their similarity to **q**



Vector model – querying

- relevance feedback supported
  - by **shifting query vector**
  - consider a **query collection**, the query vector **q** is obtained by treating all documents in the query collection as one



- the user can
  - manually shift the query vector (adjusting the weights)
  - add relevant documents from the previous search to the query collection, obtaining a shifted query vector
  - remove irrelevant documents from query collection

Vector model – implementation

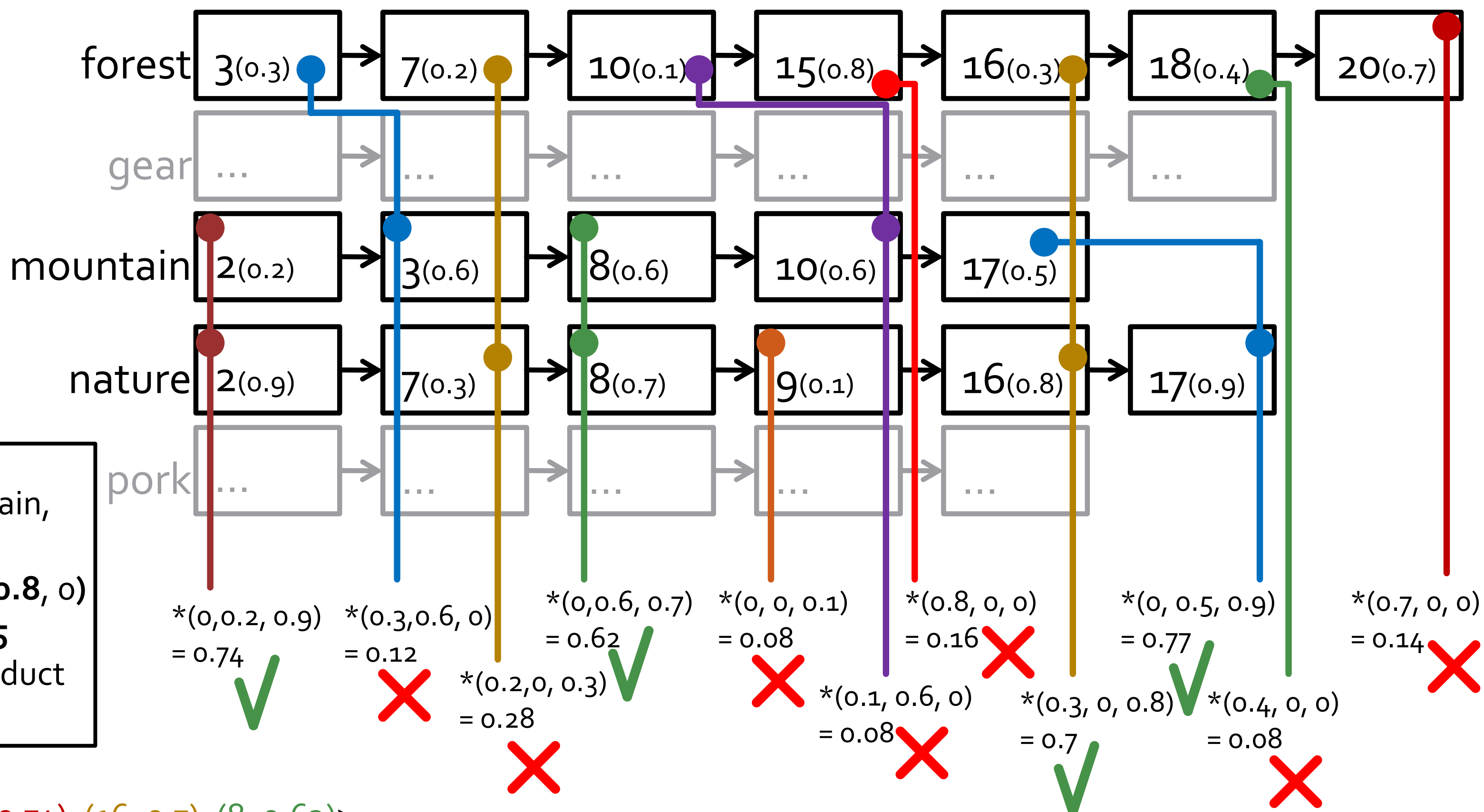
- mostly the inverted index is used
- the same data structure as for the Boolean model (previous lecture)
  - term vectors represented by inverted lists of ordered document ids the term appears in
  - additionally, the term weight is stored together with document id as DocID(term weight)

forest | 3(0.3) → 7(0.2) → 10(0.1) → 15(0.8) → 16(0.3) → 18(0.4) → 20(0.7)

- query processing similar to that of Boolean query
  - merge-sort style (aligning list cursors to the same ids)
  - but computing the cosine similarity/inner product of **q** and $\mathbf{d}_j$, not a Boolean expression

Vector model – implementation

- example of processing vector query using inverted index



Query:
"forest, mountain, nature"
q=(0.2, 0, **0.1**, **0.8**, 0)
threshold = 0.5
using inner product as similarity

Query result =
<(17, 0.77) , (2, 0.74), (16, 0.7), (8, 0.62)>

Vector model – implementation

- why inverted index?
  - useful also for Boolean queries for free
    - both vector and Boolean query evaluated by single index traversal
  - compact representation of sparse matrix (not storing zero weights)
  - term vectors indexed

- only small part of the matrix is accessed
  - zero weights (99% of the matrix) are skipped
  - only inverted lists of terms with nonzero weights in the query vector are accessed
    - query typically only a few keywords → a few lists

Vector model – implementation

- note that inverted index is only suitable when inner product or cosine similarity are used as the similarity
  - consider Euclidean distance and normed vector space* – the query would return the same as for cosine similarity, BUT
  - subtraction of coordinates, not multiplication!
    - hence, we cannot skip the inverted lists of terms with zero weights in the query, as they would contribute to the Euclidean distance

$$L_2(q, d_j) = \sqrt{\sum_{i=1}^{m} | w_{ij} - w_{iq} |^2} \qquad sim(q, d_j) = \sum_{i=1}^{m} w_{ij} w_{iq}$$

Euclidean (L2) distance vs. inner product

*The vector coordinates are divided by the vector size, so we obtain unitary vectors.

Vector model – pros and cons

- pros
  - simple and well-defined approach, geometric model
  - query-by-example (no need of a query expression)
  - provides partial matching (ranking)
  - effective model, efficient implementation
- cons
  - too simple queries, lack of the expressive power of Boolean query, syntax in the text is not considered
  - the geometerization (weighting and vector similarity) lacks of a strict semantic information
  - term independence assumed, cannot deal with linguistic phenomena like synonymy and homonymy

Latent semantic indexing (LSI) – motivation

- need to address the cons of vector model
  - vector model assumes too many independent low-level terms – but terms are not independent!
    - leads to lower precision/recall
    - e.g., two documents **"more people look for jobs"** and **"unemployment is on rise"** would not be ranked similar, while others **"George Bush fired his secretary"** and **"fires in Australian bush got strenghtened"** would be ranked as similar
  - consider higher-level concepts rather than low-level terms
    - groups of semantically similar terms (e.g., street, road, path)
    - also solves synonymy and homonymy
    - dimensionality reduction (much less concepts than terms)

The LSI model – the idea

- latent semantic indexing (LSI) = extension of the vector model
  - preprocessing of the term-by-document matrix A by the singular value decomposition (SVD)

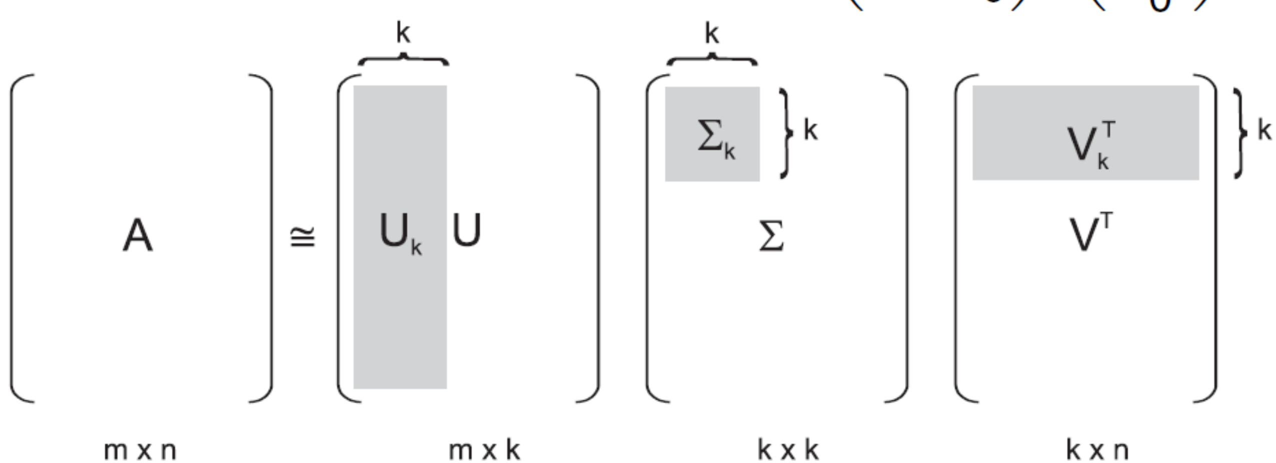
$$A = U\Sigma V^T$$

- the result is a dense *concept-by-document* matrix $\mathbf{\Sigma V^T}$
  - a document is now modeled by concepts not terms
- concept bases (vectors) in **U**
  - concept = linear combination of terms
    - statistically significant (found in the matrix A)
    - the **latent semantics** – the concept appears in multiple documents
  - concepts ordered by significance (decreasing singular values in the diagonal matrix $\mathbf{\Sigma}$) $\rightarrow$ only the first concepts in **U** are most significant

The LSI model – rank-k SVD

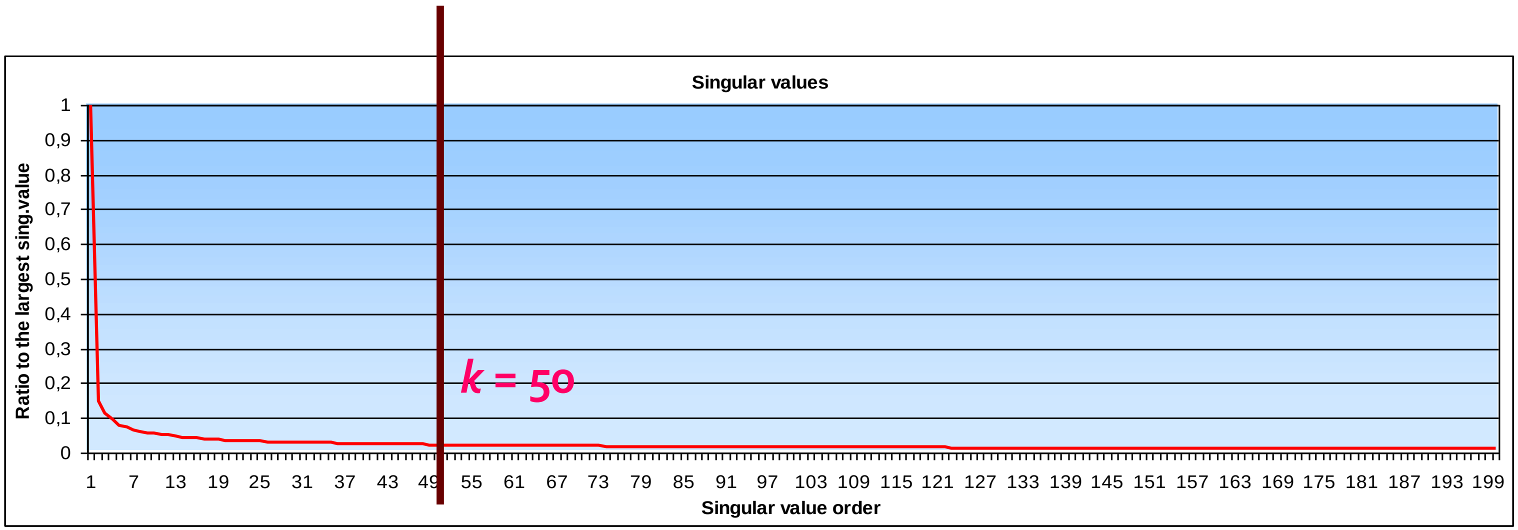
- more generally, consider SVD and a user-defined number k – the number of most significant concepts to be considered
  - we ignore the less significant concepts (beyond the k)
  - dimensionality reduction
  - more efficient algorithms of SVD decomposition

$$A = U\Sigma V^T \approx A_k = (U_k U_0) \begin{pmatrix} \Sigma_k & 0 \\ 0 & \Sigma_0 \end{pmatrix} \begin{pmatrix} V_k^T \\ V_0^T \end{pmatrix}$$



The LSI model – rank-k SVD

- usually, the singular values in Σ (significance of concepts) decrease quickly, so only several tens or hundreds is sufficient to consider
  - compare 10^1-10^3 concepts (LSI)
    to 10^4-10^5 terms (vector model)



The LSI model – the retrieval

- what is important to information retrieval
  - concept-by-document matrix, the data (document vectors)

$$D_k = \Sigma_k V_k^T$$

  - query projection
    (from the term space to the concept space)

$$q_k = U_k^T q$$

  - still cosine similarity used

The LSI model – the retrieval

- dense but low-dimensional matrix $\mathbf{D_k}$
  - dimensionality below 10^3
- also the query vector $\mathbf{q_k}$ is dense

term-by-document matrix A

document / term \	D_1	D_2	D_3	D_4	D_5
database	0	0.48	0.05	0	0.70
vector	0.23	0	0.23	0	0
index	0.43	0	0	0	0
image	0	0	0.10	0	0.54
compression	0	0	0	0	0.21
multimedia	0.12	0.52	0.62	0	0
metric	0	0	0.32	0.40	0
space	0.42	0	0	0.24	0

concept-by-document matrix D_k

document / concept \	D_1	D_2	D_3	D_4	D_5
$concept_1$	-0.21	0.48	-0.05	0.10	0.70
$concept_2$	0.23	0.20	-0.23	0.45	0
$concept_3$	-0.43	0.02	0.32	0.24	-0.06
$concept_4$	0.34	-0.01	0.10	0	0.54
$concept_5$	0.31	0.9	-0.78	0.52	0.21

The LSI model – implementation

- the inverted index not efficient data structure for LSI
  - dense concept-by-document matrix
  - dense query vector
  - processing equivalent to sequential scan of all the document vectors
- other indexing techniques needed,
  e.g., the metric access methods (later lectures)

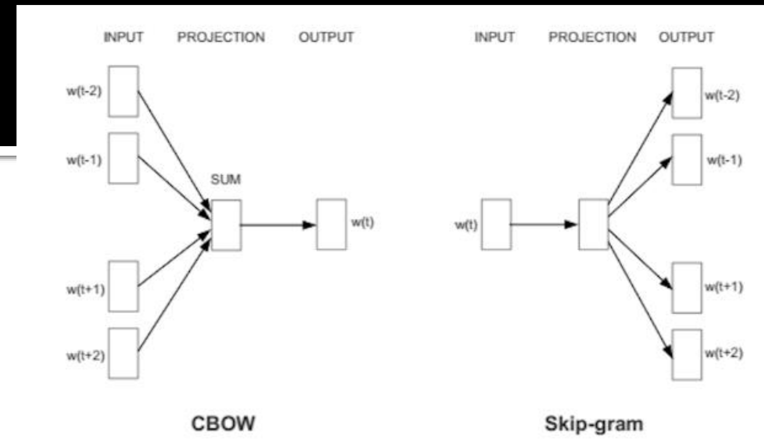
The LSI model – pros and cons

- pros
  - LSI model reveals "latent semantics" in the collection
    - the search is term-independent, it is **concept-based**
    - LSI model partially solves the problem of **synonymy** and **homonymy**
  - dimensionality reduction
    - e.g., from hundreds of thousands to several hundreds
- cons
  - concepts just statistically significant linear combinations of terms
    - not well-defined semantics (not a linguistic category)
  - dense matrix and query vector
    - inverted index not appropriate
  - expensive preprocessing of the matrix A
    - SVD algorithms have complexity $O(n^2+m^3)$

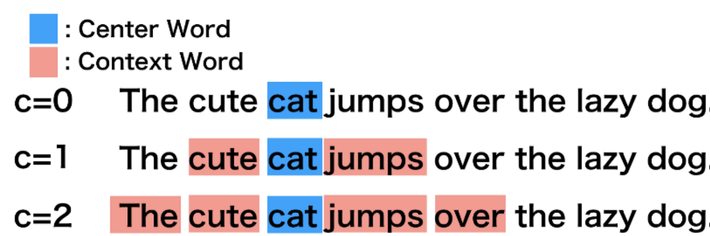
Word2vec

- distributed word representations
- machine learning approach to word embeddings
  - by Tomáš Mikolov (2013), https://github.com/tmikolov/word2vec
  - extensible to genes, code, likes/follows, playlists, social media graphs and other verbal or symbolic series
  - unsupervised, just text corpus needed for training
  - useful for NLP, search, recommendation, sentiment analysis, etc.
  - 3-layer backpropagation (gradient descent) neural network

Word2vec



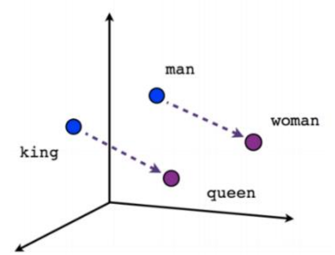
- two algorithms/network models
  - continuous bag of words (CBOW)
  - skip-grams
- N-gram context (window) for each (**center/focus**) word



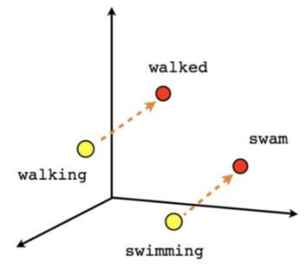
- CBOW
  - input layer: 1-of-V coding for **c** context words
    - a V-dimensional vector [0,0,0,…,1,…,0,0,0], where V is the vocabulary size
  - projection layer – continuous (target) representation of focus word
  - output layer – 1-of-V representation of focus word
- Skip-gram
  - the opposite of CBOW + order of words in context matters

Word2vec

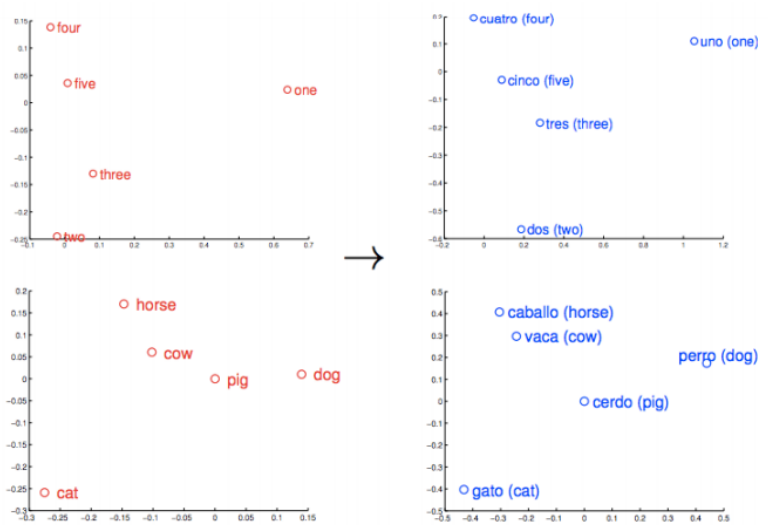
- vector arithmetic for NLP and information retrieval
  - vec("king") – vec("man") + + vec("woman") ≈ vec("queen")

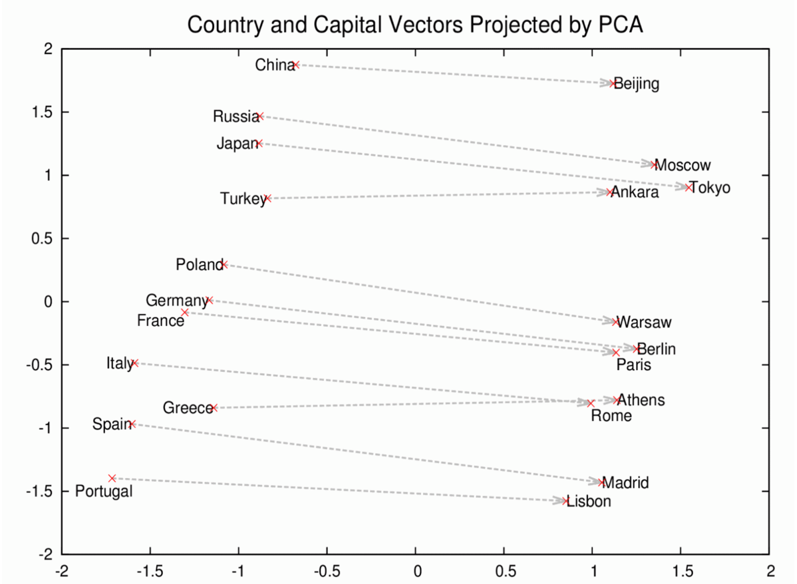


Male-Female



Verb tense





Word2vec vs. LSI (SVD)

- word representations vs. document (and word) representations
- word2vec much faster (scalable)
  - no need to build/store huge matrices, no decomposition, parallel proc.
- word2vec performs much better
  - window contexts, not entire documents
- however, SVD could be adapted to work like word2vec