Spring Semester 2020
CS-423 Distributed information systems

# Distributed information systems Summary

Author SCIPER

September 16, 2020

# Contents

# Index

# -1 TODOs

- Write sklearn big directories

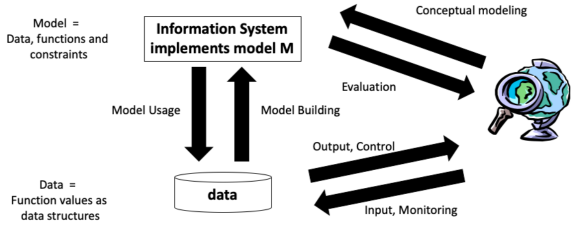


Figure 1: Information management tasks

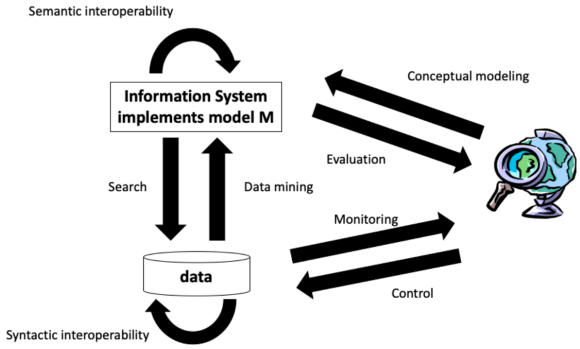


Figure 2: Information management tasks 2

# 0 Introduction

Github of the course: link.

## 0.1 LEC01 : An overview

Let's begin with some definitions:

- A computer **information system** is a system composed of people and computers that processes or interprets information. The term is also sometimes used in more restricted senses to refer to only the software used to run a computerized database or to refer to only a computer system.
- A **model** is a mathematical structure consisting of a set of constants (identifiers), functions (relations), axioms (constraints).
- Functions in models can be both computed and enumerated.
- A **data modeling language** is a language used to specify data models consisting of *Data Definition Language (DDL)* and *Data Manipulation Language (DML)*.
- A data modeling language specifies three main components: Data structures, Integrity constraints and Manipulation.
- **Data mining** is a *model building* method, not a *model usage*.
- A **monitoring task** is to build data from observing the real world. (see Figure 1 and Figure 2)
- A **distributed information systems** could use different models:
  - **Centralized**: Running on one physical node under a single authority
  - **Physically distributed** (distributed data management) among multiples information systems/applications (locality). Key issues: Where to store the data in the network? (partitioning of data, replication and caching); How to access data in the network?
  - **Heterogeneity among data** (data integration): Each information system has a different type of data and the application has a bit of each. Key issues: More data = more information? (the same real world can be modelled differently) ; Data overload ; Information starvation (data supply doesn't match data demand, models used by data provider are different from models used by customers)

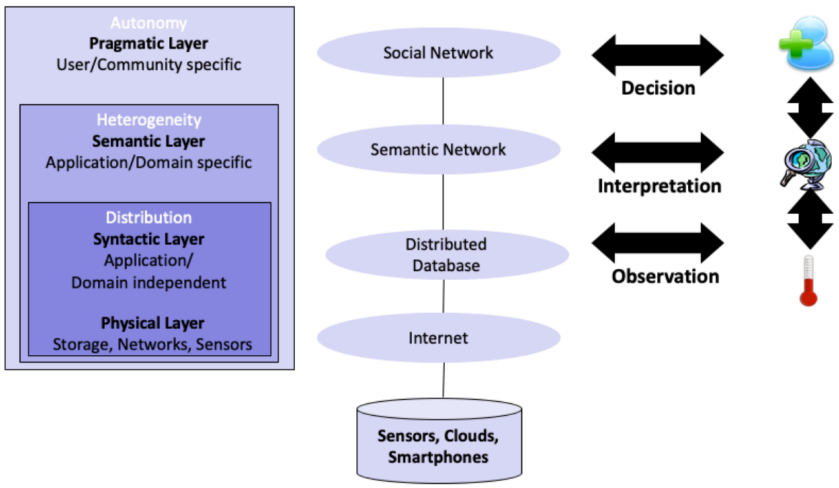


Figure 3: Refined view of DIS

– **Autonomy** (distribution of control): Independent users have to collaborate, coordinate, negotiate, to perform information management tasks. Key issues: *Trust* between users? If I reveal *quality* information to other, it increases my trust but lowers my *privacy*.

The reader could observe a refined view of the distributed information systems. (SeeFigure 3)

# 1 Information Retrieval

## 1.1 LEC02 : Basic text retrieval models

### 1.1.1 Information retrieval (IR)

- The idea of information retrieval is to find a large collection of documents that satisfy the information needs of a user. It's important to know that the documents are mostly unstructured. The first thing to do is to represents the documents (one document → d) and the user's query (q) in the same representation ($Rep_D$ and $Rep_Q$) to be able to compute the similarity between them (**feature extraction**, **query formulation**).
- Generally the features associated to a document is its content, the authors and the concepts (extracted or annotated).
- **Retrieval** produces a ranked result from a user query. That's the interpretation of the information by the system. **Browsing** let the users navigate in the information set and is a relevance feedback by the human. Both are usually connected to improve the relevance of the documents retrieved.

### 1.1.2 Text-based information retrieval

We could observe the architecture of text retrieval systems in Figure 4.

- **full-text** or **bag of words** (Basic approach): use the words that occur in a text as features for the interpretation of the content.
- Pre-processing text for text retrieval (features extraction). After each step we can simply take the result without doing the rest.

  1. Docs

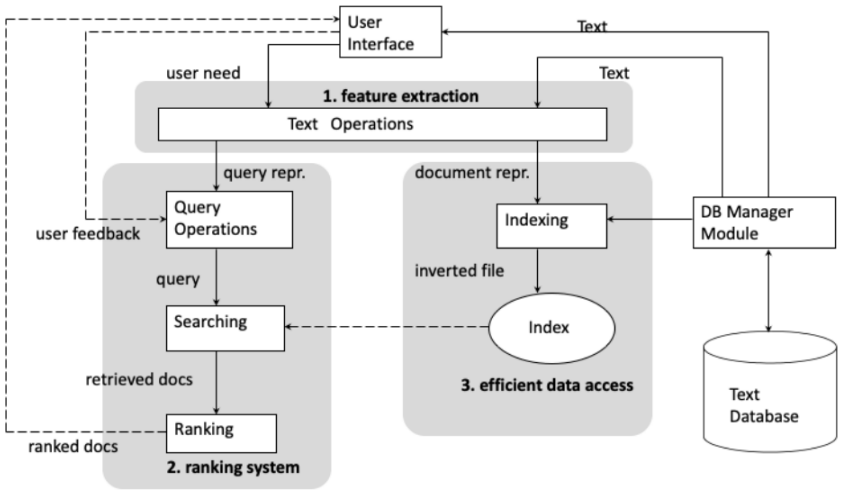


Figure 4: Architecture of text retrieval systems

2. Structure layout metadata (non-mandatory but outputs structured data)
3. Tokens (outputs the full text)
4. Stopwords
5. Stemming
6. Manual indexing (outputs only some indexed terms)

- basic concepts and notations:
  - Document: $d$
  - Query: $q$
  - Index term: a semantic unit, a word, short phrase or potentially root of a word
  - Database: $DB$. Collection of $n$ documents $d_j \in DB$, $j = 1, \ldots, n$
  - Vocabulary: $T$. Collection of $m$ index terms $k_i \in T$, $i = 1, \ldots, m$
  - Thus a document $d_j$ is represented by a set of index word $k_i$
  - $Rep_D$: The importance of an index term $k_i$ for the meaning of a document $d_j$ is represented by a weight $w_{ij} \in [0, 1]$ (if all words have equal importance, then $w_{ij} \in \{0, 1\}$); we write $d_j = (w_{1j}, \ldots, w_{mj})$. It builds a matrix of weights $w_{ij}$ (no stopwords) that indicates which terms occur in a document collection.
  - $sim$: The IR system assigns a *similarity coefficient* $sim(q, d_j)$ as a estimate for the relevance of a document $d_j \in DB$ for a query $q$.

- Boolean retrieval:
  - Retrieval language: expr::= term | (expr) | NOT expr | expr AND expr | expr OR expr
  - weights are still $w_{ij} \in \{0, 1\}$
  - "Similarity" computation (see example lecture 2 slide 29):
    1. Determine the disjunctive normal form of the query $q$ using Morgan's law. (e.g. (A AND B) OR (C AND NOT D))
    2. $Rep_Q$: for each conjunctive term $ct$ create its query weight vector $vec(ct) = (w_1, \ldots, w_m)$ with $w_i = 1$ if $k_i \in ct$, $w_i = -1$ if $k_i \notin ct$ else $W_i = 0$

3. A match is established if the document vector contains all the terms of the query vector in the correct form. (i.e. $sim(d_j, q) = 1$ and $vec(ct)$ matches $d_j$ if ($w_i = 1$ AND $w_{ij} = 1$) OR ($w_i = -1$ AND $w_{ij} = 0$))

– Problems: no ranking, query difficult to formulate, no error tolerance, queries return many or none results.

### 1.1.3 Vector space retrieval

Key idea:

- Use "free text" queries.
- represents both document and query by a weight vector of $m$-dimensions with **non-binary** weights.
- Compute the distance query-document in the m-dimensional space.

Let's see the calculation for that:

$$\vec{d_j} = (w_{1j}, w_{2j}, \ldots, w_{mj}), w_{ij} > 0 \text{ if } k_j \in d_j$$

$$\vec{q} = (w_{1q}, w_{2q}, \ldots, w_{mq}), w_{iq} \geq 0$$

$$sim(\vec{q}, \vec{d_j}) = cos(\theta) = \frac{\vec{d_j} \bullet \vec{q}}{|\vec{d_j}| \cdot |\vec{q}|} = \frac{\sum_{i=1}^{m} w_{ij} w_{iq}}{|\vec{d_j}| \cdot |\vec{q}|}, \ 0 \leq sim(\vec{q}, \vec{d_j}) \leq 1$$

$$|\nu| = \sqrt{\sum_{i=1}^{m} \nu^2}$$

Properties:

- Ranking of documents according to similarity value
- Documents can be retrieved even if they contains only some (i.e. not every) query words.
- Simple and fast to compute.

**Term-frequency**(TF): $freq(i, j)$ of the keyword $k_i$ in the document $d_j$. Recall that $T$ is the vocabulary.

$$tf(i, j) = \frac{freq(i, j)}{\max_{k \in T} freq(k, j)}$$

**Inverse-document-frequency**(IDF): how frequent a term is in the document collection of size $n$ (measure of distinctiveness). It can be interpreted as the amount of information associated with the term $k_i$. That's a value associated to each term of the vocabulary (i.e. we can compute it once at the beginning). Inverse document frequency of term $k_i$ (recall that $n_i$ is the number of document that contains the term $i$):

$$idf(i) = log(\frac{n}{n_i}) \in [0, log(n)]$$

**TF-IDF**: Term-weight $\rightarrow w_{ij} = tf(i, j) \cdot idf(i)$
**Query weights**: same idea for the query.

$$w_{iq} = \frac{freq(i, q)}{\max_{k \in T} freq(k, q)} log(\frac{n}{n_i})$$

**Length normalization**: When computing cosine similarity, document vectors are normalized. Which implies that shorter documents will be favored. Original scheme (see Figure 5):

$$\frac{tf \cdot idf}{(1 - s) \cdot pivot + s \cdot n}$$

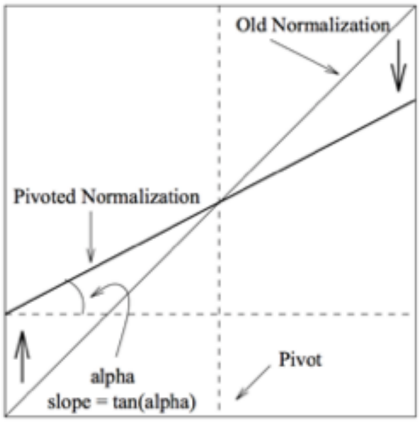


Figure 5: Cosine normalization factor

| Term frequency | | Document frequency | | Normalization | |
|---|---|---|---|---|---|
| n (natural) | $\mathrm{tf}_{t,d}$ | n (no) | 1 | n (none) | 1 |
| l (logarithm) | $1+\log(\mathrm{tf}_{t,d})$ | t (idf) | $\log \frac{N}{\mathrm{df}_t}$ | c (cosine) | $\frac{1}{\sqrt{w_1^2+w_2^2+\ldots+w_M^2}}$ |
| a (augmented) | $0.5+\frac{0.5\times\mathrm{tf}_{t,d}}{\max_t(\mathrm{tf}_{t,d})}$ | p (prob idf) | $\max\{0,\log\frac{N-\mathrm{df}_t}{\mathrm{df}_t}\}$ | u (pivoted unique) | $1/u$ |
| b (boolean) | $\begin{cases}1 & \text{if } \mathrm{tf}_{t,d}>0\\ 0 & \text{otherwise}\end{cases}$ | | | b (byte size) | $1/CharLength^{\alpha}$, $\alpha<1$ |
| L (log ave) | $\frac{1+\log(\mathrm{tf}_{t,d})}{1+\log(\mathrm{ave}_{t\in d}(\mathrm{tf}_{t,d}))}$ | | | | |

Figure 6: Variants of vector space retrieval models

- $s$ = slope
- $n$ = old normalization factor
- Advantage: Better retrieval performance
- Disadvantages: Normalization is collection specific ; Needs to be empirically found

Advantages and disadvantages of vector space retrieval models:

- 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 degree of similarity to the query.
- Disadvantages:
  - assumes independence of index terms
  - no theoretical justifications why it works

### 1.1.4 Evaluating information retrieval

**Recall**: The fraction of relevant documents retrieved from the set of total relevant documents collection-wide. How large a fraction of the expected results is actually found. It can be optimized by simply returning the whole document collection. (See Table 1). $R = \frac{tp}{tp+fn} = P(retrieved|relevant)$

| | relevant | non-relevant |
|---|---|---|
| retrieved | true positive (tp) | false positive (fp) |
| not retrieved | false negative (fn) | true negative (tn) |

Table 1

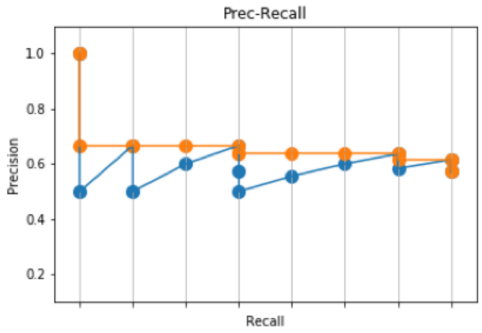


Figure 7: R N R N R R N N R R R N R N R R ; Blue: precision-recall plot ; Orange: interpolated precision

**Precision**: The fraction of relevant documents retrieved from the total number retrieved. How many of the results returned are actually relevant. It can be optimized by returning only very few results. (See Table 1). $P = \frac{tp}{tp+fp} = P(relevant|retrieved)$.

**Note**:*High recall hurts precision and vice-versa!* The higher the precision for a specific recall, the better the information retrieval system.

**Important note**: Both measures evaluate an **unranked** result set. All elements of the result are considered as equally important.

**Note**: If a system ranks the results according to relevance the user can control the relation between recall and precision by selecting a threshold of how many results he/she inspects.

**F-measure**: weighted harmonic mean. F1: balanced F-measure, with $\alpha = \frac{1}{2} \implies F1 = \frac{2PR}{P+R}$. Larger values of alpha emphasize the importance of precision and smaller ones the importance of recall.

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}}, \alpha \in [0,1]$$

**Interpolated precision**: (See Figure 7)

$$P_{int}(R) = \max_{R' \geq R} P(R')$$

**Mean average precision** (MAP): Given a set of queries $Q$. For each $q_j \in Q$ the set of relevant documents $\{d_1, \ldots, d_{m_j}\}$. $R_{jk}$ is the top-k documents for query $q_j$. $P_{int}(R_{jk})$ interpolated precision of result $R_{jk}$. It's a robust measure for evaluating the quality of a ranked retrieval system for a query collection $Q$. When a relevant document is not retrieved at all, the precision value in the MAP is 0. It is a sum of the averages and at the end we normalize by $\frac{1}{|Q|}$ (e.g. lecture 2 slide 58).

$$MAP(Q) = \frac{1}{|Q|}\sum_{j=1}^{|Q|}\frac{1}{m_j}\sum_{k=1}^{m_j}P_{int}(R_{jk})$$

**ROC curve**: It relates specificity (on the x-axis) to recall (on the y-axis). Specificity measures the fraction of true negatives that are detected in proportion to all negatives. Thus 1−specificity is the rate of false positives that have been retrieved, the so-called **false positive**

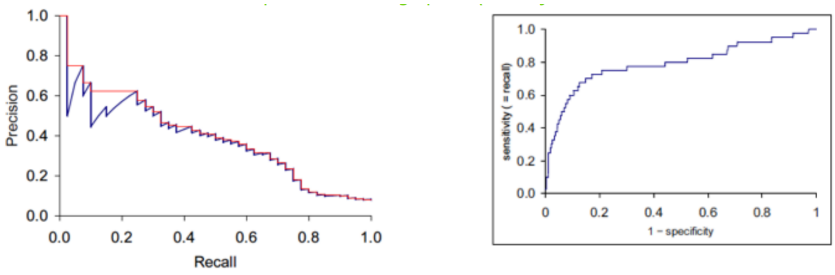


Figure 8: ROC curve

**rate (FPR)**. The steeper the curve rises at the beginning, the better. The larger the area under the curve, the better. (See Figure 8)

$$\text{Specificity } S: 1 - S = \frac{fp}{fp + tn} = P(retrieved|notrelevant)$$

Specificity gives information about how many of the true negatives have been retrieved as false positives.
Example: *When the* $R = 0.5$ *then* $1 - S = 0.1$. *This implies that* $S = 0.9$. *Then we can conclude that* $fp = \frac{1}{9}tn$, *or in other words when retrieving half of the relevant documents, then we have also retrieved about* 10% *of the non-relevant documents.*

## 1.2 LEC03 : Information Retrieval Indexing

Previously we covered the first two parts of Figure 4 (i.e. feature extraction & ranking system). Let's talk about the third one (i.e. efficient data access). The problem is that text retrieval algorithms need to find words in documents efficiently (given index term $k_i$, find documents $d_j$. e.g. "application" $\rightarrow d_2, d_{13}$).

### 1.2.1 Indexing for information retrieval

An **inverted file** is a word-oriented mechanism for indexing a text collection in order to speed up the term search task. Optimized for search on relatively static text collections.

- The notation for an *inverted list* $l_k$ for a term $k$ is $l_k = [f_k : d_{i_1}, \ldots, d_{i_{f_k}}]$ where $f_k$ is the number of document in which $k$ occurs and $d_{i_1}, \ldots, d_{i_{f_k}}$ is the list of document identifiers of documents containing $k$.
- An *inverted file* is a lexicographically ordered sequence of inverted lists. $IF = [i, k_i, l_{k_i}]$ with $i = 1, \ldots, m$
- *Example:*

| Term | nb occurrences | index of document |
|---|---|---|
| Algorithms | 3 | 3, 5, 7 |
| Application | 2 | 3, 17 |
| ... | ... | ... |
| Methods | 2 | 5, 8 |
| Nonlinear | 4 | 1, 12, 13, 17 |
| ... | ... | ... |

- You can observe the physical organisation of inverted files at Figure 9. (Note: when indexing a document collection using a inverted file, the main space requirement is implied by the posting file.)

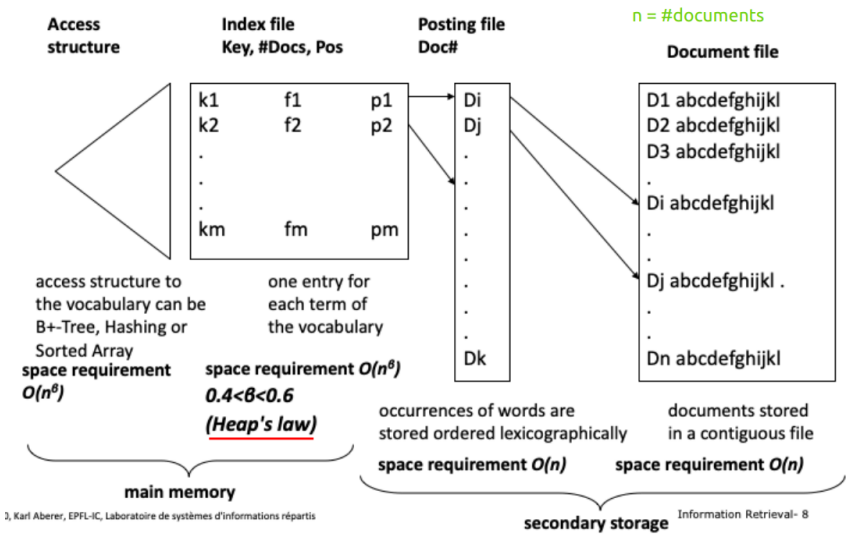


Figure 9: Physical organization of inverted files

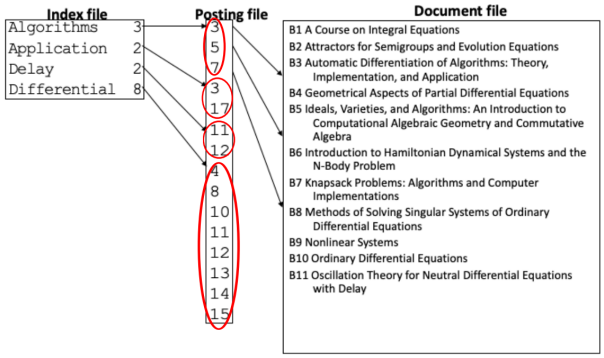


Figure 10: Example of searching the inverted file

- **Heap's Law** is an empirical law that describe the relation between the size of a collection and the size of its vocabulary. $m = kn^{\beta}$ where we have the following typical values $\beta \approx \frac{1}{2}$, $30 < k < 100$ and $n$ is the number of documents. *For example, a document collection of size* $n = 10^6$ *could have approximate* $m = 100 \cdot 10^3 = 10^5$ *index terms.*
- Searching the inverted file consists of 3 steps (e.g. Figure 10):
  1. *Vocabulary search*: search the words from the query in the index file.
  2. *Retrieval occurrences*: the lists of occurrences of all words are retrieved from the posting file (a **posting** indicates the occurrence of a term in a document).
  3. *Manipulation of occurrences*: the occurrences are processed in the document file to process the query.

  Let's detail each step.

  1. The vocabulary is kept in an ordered data structure (e.g. trie or sorted array) and each word store a list of its occurrences. Each word of the text is read sequentially and search in the vocabulary. If not in the vocabulary, then add it with an empty list of occurrence. Then add the word position to the end of this list.
  2. storage phase (once the text is exhausted). The list of occurrence is written contiguously to the disk (it creates the posting file). The vocabulary is stored in lexicographically order (index file) in main memory together with a pointer for each word to its list in the posting file. (cost: $O(n)$).

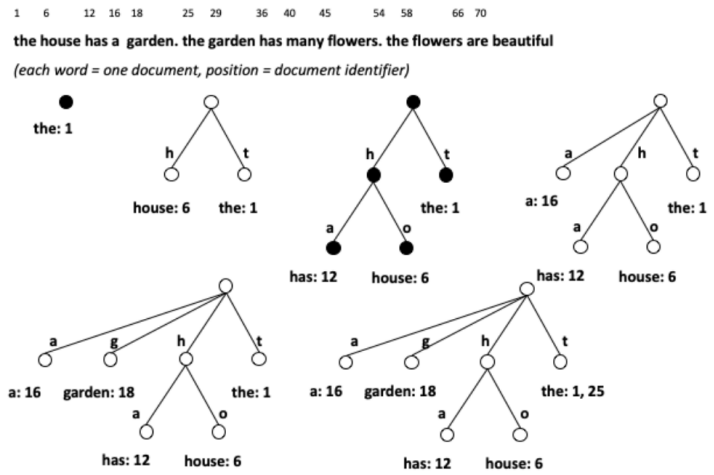


Figure 11: build an inverted file: step 1

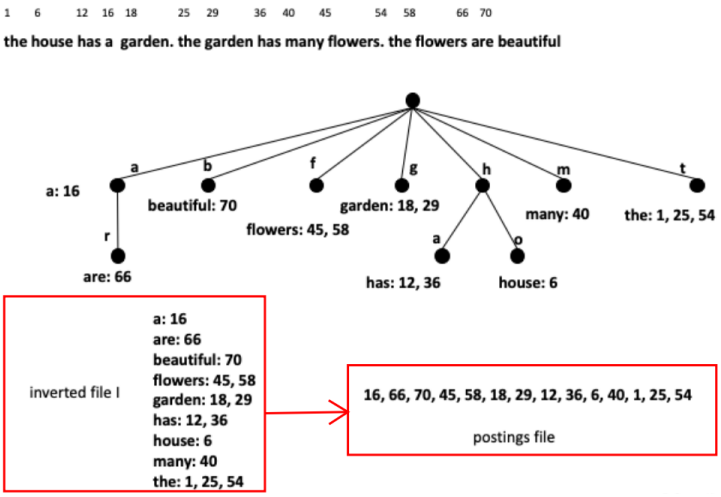


Figure 12: build an inverted file: step 2

3. what write here?

- The index construction: The reader can see an example at Figure 11, Figure 12 and Figure 13. In this example we consider each word of the text as a separate document identified by its position (for space limitations). Using a trie in index construction has 3 main advantages:
  - Helps to quickly find words that have been seen before.
  - Helps to quickly decide whether a word has not been seen before.
  - Helps to maintain the lexicographic order of words seen in the documents.

  Index construction in practice: On a single node machine the index construction will be inefficient or impossible if the size of the trie structure with the associated posting lists exceeds the main memory space $\Longrightarrow$ *while the document collection is sequentially traversed, partial indices are written to the disk whenever the main memory is full.* This results in a number of partial indices, indexing consecutive partitions of the text. *In a second phase (**index merging**) the partial indices need to be merged into one index.*
- Index merging: Blocked sort-based indexing (BSBI). See Figure 14. Total cost: $O(nlog_2(\frac{n}{M}))$ where $M$ is the size of the memory. The additional cost of merging as compared to the

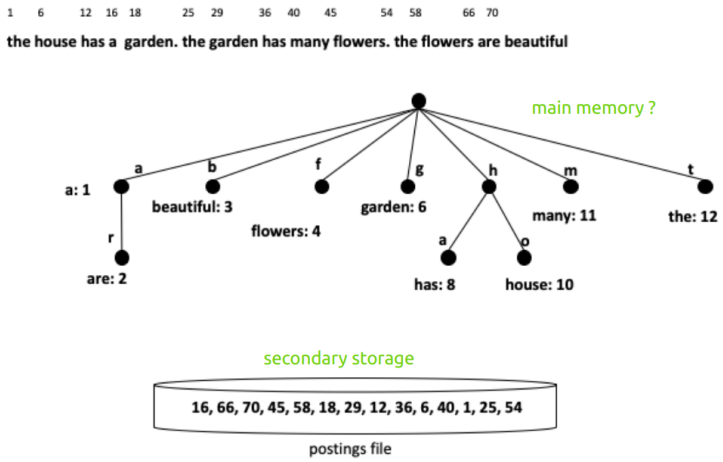


Figure 13: build an inverted file: step 3

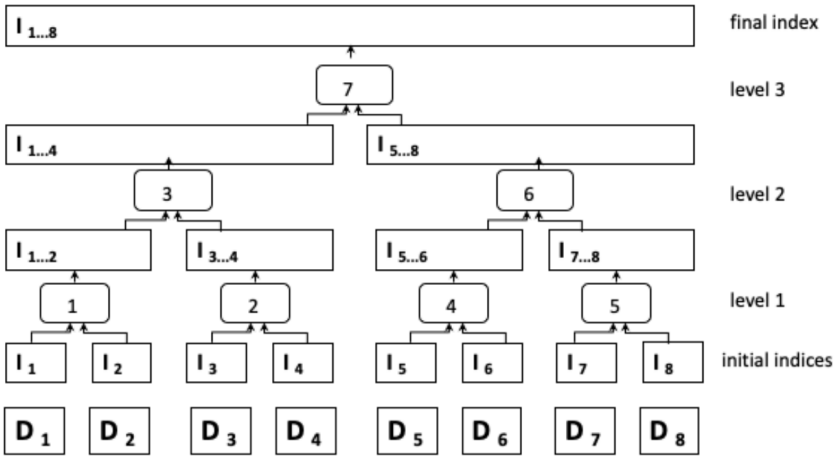


Figure 14: BSBI

purely main memory based construction of inverted files is a factor of $O(log_2(\frac{n}{M}))$.

- The posting file has the by far largest space requirements. An important factor determining the size of an inverted file is the addressing granularity used. The addressing granularity determines of how exactly positions of index terms are recorded in the posting file. There exist three main options:
  - Exact word position
  - Occurrence within a document
  - Occurrence within an arbitrary sized block = equally sized partitions of the document file spanning probably multiple documents

  The larger the granularity, the fewer entries in the posting file but the bigger post-processing. Coarser granularities lead to a reduction of the index size because: reduction in pointer size (e.g. from 4 Bytes for word addressing to 1 Byte with block addressing) ; lower number of occurrences.
  An improvement has been seen in class: Index compression at lecture 3 slide 24.

- Map-reduce:
  - We work on key-value pairs $(k, v)$ and have a map and a reduce functions.

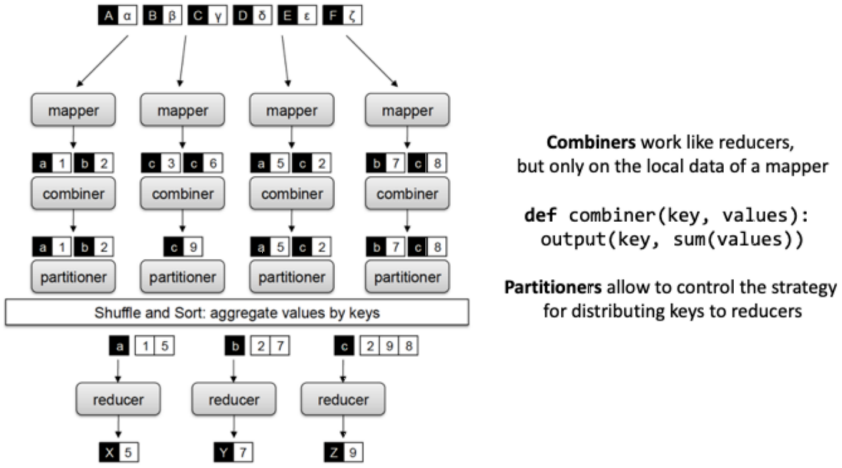


Figure 15: map-reduce program model

– *Example: basic word counter program. See Figure 15*

```
def mapper(document, line):
    for word in line.split(): output(word, 1)

def reducer(key, values):
    output(key, sum(values))
```

– The programmer controls:
  * Key-value data structures (can be complex)
  * Maintenance of state in mappers and reducers
  * Sort order of intermediate key-value pairs
  * Partitioning scheme on the key space.
– The map-reduce platform controls:
  * where the mappers and reducers run
  * when a mapper and reducer starts and terminates
  * which input data is assigned to a specific mapper
  * which intermediate key-value pairs are processed by a specific reducer
– Construction of inverted file using map-reduce. Mappers extracts postings from documents and provide them to the reducers. Reducers aggregate posting lists

```
def mapper(doc, text):
    f = {}
    for word in text.split(): f[word] += 1
    for word in f.keys(): output(word, (doc, f[word]))

def reducer(key, postings):
    p = []
    for d, f in postings: p.append((d, f))
    p.sort()
    output(key, p)
```

### 1.2.2 Distributed retrieval

The idea is that we could have postings lists distributed among different nodes. Do we have to transfer the complete posting list to identify the top-k documents? We will try to answer that

| query term 1 | | query term 2 | |
|---|---|---|---|
| doc | tf-idf | doc | tf-idf |
| d1 | 0.9 | d6 | 0.81 |
| d4 | 0.82 | d2 | 0.7 |
| d3 | 0.8 | d5 | 0.66 |
| d5 | 0.65 | d1 | 0.45 |
| ... | ... | ... | ... |
| d6 | 0.51 | d3 | 0.33 |
| d2 | 0.01 | d7 | 0.15 |
| d7 | 0.0 | d4 | 0.0 |

| doc | query term 1 | query term 2 | sum |
|---|---|---|---|
| d1 | 0.9 | 0.45 | 1.35 |
| d6 | | 0.81 | 0.81 |
| d4 | 0.82 | | 0.82 |
| d2 | | 0.7 | 0.7 |
| d3 | 0.8 | | 0.8 |
| d5 | 0.65 | 0.66 | 1.31 |

Table 2: Example Fagin after step 1 finished

| query term 1 | | query term 2 | |
|---|---|---|---|
| doc | tf-idf | doc | tf-idf |
| d1 | 0.9 | d6 | 0.81 |
| d4 | 0.82 | d2 | 0.7 |
| d3 | 0.8 | d5 | 0.66 |
| d5 | 0.65 | d1 | 0.45 |
| ... | ... | ... | ... |
| d6 | 0.51 | d3 | 0.33 |
| d2 | 0.01 | d7 | 0.15 |
| d7 | 0.0 | d4 | 0.0 |

| doc | query term 1 | query term 2 | sum |
|---|---|---|---|
| d1 | 0.9 | 0.45 | 1.35 |
| d6 | 0.51 | 0.81 | 1.32 |
| d4 | 0.82 | 0.0 | 0.82 |
| d2 | 0.01 | 0.7 | 0.8 |
| d3 | 0.8 | 0.33 | 1.13 |
| d5 | 0.65 | 0.66 | 1.31 |

Table 3: Example Fagin after all steps finished

question.
**Fagin's algorithm**: Entries in the posting list are sorted according to the tf-idf weights.

1. scan in parallel all lists in round-robin till $k$ documents are detected that occurs in all lists.
2. Lookup the missing weights for documents that have not been seen in all lists.
3. Sum and select the top-k elements.

*Example: finding the top-2 (implies $k = 2$) elements for a two-term query* At the first step we simply fill the "temporary" table with tf-idf scores. We stopped because we search for $k = 2$ and we have two documents in which all term of the query appears (green cells) (see Table 2). After that we look for the tf-idf score that are missing in the "temporary" table and compare the sum of the tf-idf to get the top-k documents (orange cells) (see Table 3).
**Complexity**:

- $O(\sqrt{(kn)})$ entries are read in each list for $n$ documents.
- Assuming that entries are uncorrelated.
- Improves if they are positively correlated.

### 1.2.3 Query expansion

If the user query does not contain any relevant term, a corresponding relevant document will not show up in the result. How to add documents really close to the query (query: "car", we will return documents with "automobile" , ...). There exists two main approaches:

- **Local approach**: use information from current query results (user relevance feedback)

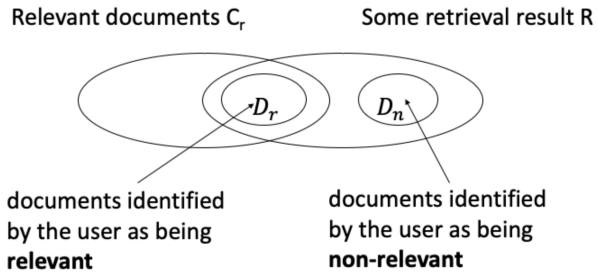


Figure 16: Feedback from users

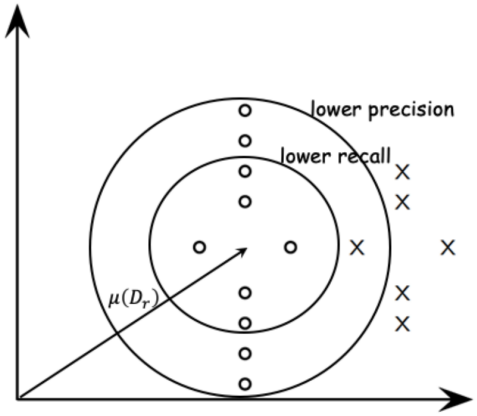


Figure 17: Illustration of Rocchio's algorithm, problem

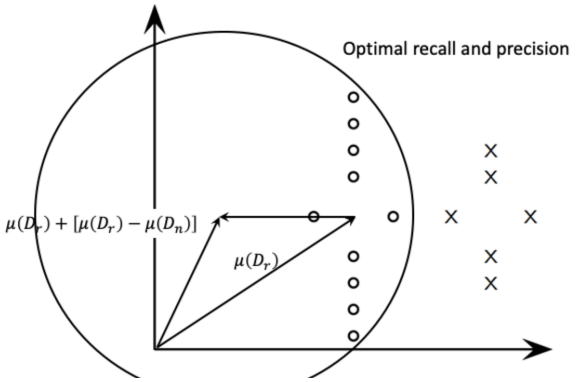


Figure 18: Illustration of Rocchio's algorithm, solution

- **Global approach**: use information from a document collection (query expansion)

1. **User relevance feedback**. In fact, after the results have been provided to the user, it sends feedback on the results to the ranking system. The ranking systems adapts the results accordingly. The advantages of such an approach are the following:

   - The user is not involved in query formulation, but just points to interesting data items.
   - The search task can be split up in smaller steps.
   - The search task becomes a process converging to the desired result.

   **Rocchio's algorithm**: find a query that optimally separates relevant from non-relevant documents (see Figure 16). Relevant documents are marked by circles and non-relevant one with crosses. Assume a vector space of only 2 dimensions. We see that we cannot achieves optimal precision and recall at the same time while considering the centroid of the relevant documents a potential search query (see Figure 17). But we can with the Rocchio's algorithm (see Figure 18).

$$\vec{q}_{opt} = \underset{\vec{q}}{\operatorname{argmax}}[sim(\vec{q}, \mu(D_r)) - sim(\vec{q}, \mu(D_n))]$$

   where the centroid of a document is

$$\mu(D) = \frac{1}{|D|} \sum_{d \in D} \vec{d}$$

   We have to keep in mind that in practice the best we can do is to approximate the optimal

solution because a perfect solution like in Figure 18 is often impossible to build. The approximation scheme is called **SMART**.
If users identify some relevant document $D_r$ from the result set $R$ of a retrieval query $q$. We have

- Assume all elements in $R \setminus D_r$ are not relevant (i.e. $D_n = R \setminus D_r$)
- Modify the query to approximate theoretically optimal query with

$$\vec{q}_{approx} = \alpha\vec{q} + \frac{\beta}{|D_r|}\sum_{\vec{d}_j \in D_r}\vec{d}_j - \frac{\gamma}{|R \setminus D_r|}\sum_{\vec{d}_j \notin D_r}\vec{d}_j$$

- $\alpha$, $\beta$, $\gamma$ are tuning parameters with $\alpha, \beta, \gamma \geq 0$
- Negative term weights are ignored. This means, for the negative term weights in $\vec{q}_{approx}$, we set them to be 0.
- Usually the original query vector is maintained ($\alpha = 1$) and the weight given to the modification using the non-relevant centroid is kept lower than the one using relevant because that's just an assumption. (e.g. $\beta = 0.75$ and $\gamma = 0.25$)

The reader can see an example in lecture 3 slide 56. If the reader does not provide any feedback, we can take the top-k documents as the relevant one and run SMART anyway (keep in mind that it can horribly failed).

2. **Query expansion**. This method uses a global *query-independent* resource (**thesaurus** is a database that contains (near-) synonyms):
   (a) Manually edited thesaurus:
      - Expensive to maintain
      - Used mainly in science and engineering
   (b) Automatically extracted thesaurus, using term co-occurrence: Attempt to generate a thesaurus automatically by analyzing the distribution of words in documents. There are two *different* definitions for the similarity between two words:
      i. **Word embeddings**: *Two words are similar if they co-occur with similar words.* (e.g. "switzerland" ≈ "austria" because both appears with words such as "national", "election", ...)
      ii. **Information extraction**: *Two words are similar if they occur in a given grammatical relation with the same word.* (e.g. "live in *", "travel to *", "size of *" are all phrases in which "switzerland" and "austria" can occur)
   (c) Query logs:
      - Example 1: After searching "Obama", users search "Obama president". Therefore "president" might be a good expansion
      - Example 2: User A accesses "epfl.ch" after searching "Aebischer". User B accesses "epfl.ch" after searching "Vetterli". Therefore, "Vetterli" might be a good expansion for the query "Aebischer" and vice-versa.

## 1.3 LEC04 : Advanced Retrieval Methods

### 1.3.1 Probabilistic information retrieval

Probabilistic IR models attempt to directly model relevance as probability.

**Query likelihood model:** Given a query $q$, determine $P(d|q)$ that document $d$ is relevant to query $q$. $P(d)$, the probability of a document occurring is uniform across a collection. $P(q)$

is the same for all queries. By Bayes' rule $P(d|q)$ can be derived from $P(q|d)$ (called the **query likelihood**).

Let's try to determine $P(q|d)$. Assume each document $d$ is generated by a **language model** $M_d$. Then $P(q|d)$ can be interpreted as the probability that the query $q$ was generated by the language model $M_d$. The idea is: if a query is relevant to a document, it should have been produced by the same language model as the document. Using this argument, the query likelihood corresponds to the probability that the query has been produced by the same language model as the document.

A *language model* is a mechanism that generates the words of the language. We will use a specific case of probabilistic language model: An automaton with *single* state $Q$ (a unigram) with $P(STOP|Q) = \alpha$ and if you go to $Q$ you select a word $w$ with probability $P(w) = p_w$. Using that, the retrieval becomes the problem of computing $P(q|M_d)$ for all the documents $d$ (one language model is generated per document).

**Maximum likelihood estimation** (MLE) of probability under unigram model:

$$\widehat{P}_{mle}(t|M_d) = \frac{tf_{t,d}}{L_d}$$

where $tf_{t,d}$ is the term frequency of $t$ in $d$ and $L_d$ is the length of document $d$ (#terms). Assuming the independence we use the following model:

$$\widehat{P}(q|M_d) = \prod_{t \in q} \widehat{P}_{mle}(t|M_d)$$

*Example: "Information retrieval is the task of finding the document satisfying the information needs of the users."* $\implies P(the|M_d) = \frac{1}{4}$ *and* $P(information|M_d) = \frac{1}{8}$

*Example:* $d =$ *"information retrieval and search"* $\implies P(information\ search|M_d) = \frac{1}{4^2} < \frac{1}{4} = P(information|M_d)$

If $t \in q$ but $t \notin M_d$ then $\widehat{P}(q|M_d) = 0$. We need to give a non-zero probability to unseen terms. **Smoothed estimate**

$$\widehat{P}(t|d) = \lambda \widehat{P}_{mle}(t|M_d) + (1 - \lambda)\widehat{P}_{mle}(t|M_c)$$

where $M_c$ is the language model of the whole collection (i.e. put the content of every documents into a single giant document) and $\lambda$ is a tuning parameter (can be query dependent e.g. query size). We obtain

$$P(d|q) \propto P(d) \prod_{t \in q} ((1 - \lambda)P(t|M_c) + \lambda P(t|M_d))$$

There is an example lecture 4 slide 14.

In class we have seen that probabilistic retrieval (compared to vector space retrieval) improves precision for higher values of recall. Finally we will use vector space retrieval when a quick and simple solution is needed (no tuning needed) but a probabilistic approach will achieve better performance.

### 1.3.2 Latent semantic indexing

Vector space based retrieval are vague and noisy. They are based on index terms not concepts. The user information need is more related to concepts and ideas than index words. Vector space retrieval are bad in two situations:

- *Synonymy*: Different terms refer to the same concept (e.g. car-automobile) (result:poor recall)
- *Homonymy*: Same terms refer to different concept (e.g. apple (mobile-fruit)) (result: poor precision)

The idea is to map the documents and queries to a lower-dimensional space that refer to higher-level concepts. Instead of linking terms directly to documents ($t_i \to d_j$) link terms to concepts to documents ($t_i \to c_k \to d_j$). A concept can be linked to multiple terms and to multiple documents. Then we can express documents and queries by concept vector (counting # of concept terms). The similarity is computed by scalar product of *normalized* concept vectors.

*Example: if* $d_1 = (4, 3, 3, 1) \to (4/11, 3/11, 3/11, 1/11)$ *and* $d_2 = (3, 1, 3, 3) \to (0.3, 0.1, 0.3, 0.3)$ *then* $sim(d_1, d_2) = 0.245$ Bullshit!

How to identify and compute concepts? Consider the **term-document** matrix.

- Let $M_{ij}$ be a term-document matrix with $m$ rows(terms) and $n$ columns(documents)
- to each element of this matrix is assigned a weight $w_{ij}$ with $t_i$ and $d_j$
- The weight $w_{ij}$ can be based on a tf-idf weighting scheme.

The ranking is the result of computing the product of a query vector $q$ with the term-document matrix $M$ (rank $= M^t \cdot q$) where all columns in $M$ and $q$ are normalized to 1.

**Singular value decomposition** (SVD): Represent matrix $M$ as $M = K \cdot S \cdot D^t$

- $K$ and $D$ are matrices with orthonormal columns (e.g. $K \cdot K^t = I = D \cdot D^t$).
- $K$ is the m x r matrix of eigenvectors derived from $M \cdot M^t$. It represents terms in concepts space (or inversely).
- $D$ is the n x r matrix of eigenvectors derived from $M^t \cdot M$. It represents documents in concepts space (or inversely).
- $S$ is an r x r diagonal matrix of the singular values sorted in decreasing order where $r = min(m, n)$ (i.e. rank$(M)$).
- The decomposition always exists and is unique (up to sign).
- The complexity of the construction is $O(n^3)$ if $m \leq n$. (There exists some approximation technique instead of perfect result).

Interpretation of SVD: We can write $M$ as a sum of outer vector products.

$$M = \sum_{i=1}^{r} s_i k_i \bigotimes d_i^t$$

where $s_i$ are decreasing. If we take only the largest ones we obtain a good approximation of $M$ (least square approximation). We can interpret the axes of the hyper-ellipsoid E as the dimensions of the concept space.

**Latent Semantic Indexing** (LSI): In the matrix $S$ take only the $s \leq r$ largest singular values (keep the corresponding columns in $K$ and $D$). Call the result $M_s$. The tuning of $s$ is crucial. We obtain the same scheme as before while replacing $r$ by $s$ in the size of matrices.

To answer queries, documents can be compared by computing cosine similarity in the concept space (i.e. columns $(D_s^t)_i$ and $(D_s^t)_j$ in matrix $D_s^t$). A query $q$ is treated like one further document (added as an additional column to $M$ (and to $D_s^t$ but not to $K_s$) ; the same transformation is applied to this column as for mapping $M$ to $D$).

Mapping of $M$ to $D$: $D = M^t \cdot K \cdot S^{-1}$ and apply same transformation to $q$: $q^* = q^t \cdot K_s \cdot S_s^{-1}$. Then

$$sim(q^*, d_i) = \frac{q^* \cdot (D_s^t)_i}{|q^*||(D_s^t)_i|}$$

Using that decomposition we are able to plot the terms and documents in a $s$-dimensional concept space.

Advantages and disadvantages of LSI:

+ It allows reducing the complexity of the underlying concept representation.

+ Facilitates interfacing with the user

- Computationally expensive

- Poor statistical explanation

**Latent Dirichlet Allocation** (LDA): That's an alternative to the LSI. The idea is to assume a document collection is (randomly) generated from a known set of topics (probabilistic generative model) (i.e. For each document, choose a mixture of topics (with weights). For every word position sample a topic from the topic mixture given the weights. For every word position sample a word from the chosen topic). The approach is to invert the process: given a document collection reconstruct the topic model. The main advantage is that topics in LDA are interpretable unlike the arbitrary dimensions of LSI. This method is currently considered as the state-of-the-art method for topic identification.

## 1.4 LEC05 : Link-Based Ranking

### 1.4.1 Word embeddings

The neighborhood of a word expresses a lot about its meaning. The **context** $C(w)$ of a word $w$ is the words that are around it. Two words are considered as similar when they have similar contexts (it captures syntactic (e.g. king-kings) and semantic (e.g. king-queen) similarity). That is the main difference to methods based on document co-occurrence like LSI. A second difference is to distinguish between a context and as a member of context. For example, it is very unlikely the word "dog" would have in its context a second occurrence of the word "dog". Thus dog as a "context word" needs to be treated differently from dog as the word of interest. This similarity-based representation is implicitly used with the term-document matrix $M$.

The idea of word embeddings is to model how likely a word and a context occur together. The basic idea is to map both words and context words into the same low-dimensional space. Interpret the vector product as a measure for that. Due to projection in low-dimensional space, similar semantically words and contexts should be close.

In summary we will map a word $w_i$ with $\vec{w_i} = W^{(w)}w_i$. $w_i$ is a vector with 1 at position $i$ and 0 otherwise. Thus $\vec{w_i}$ is the column $i$ in $W^{(w)}$. Same idea for the mapping of a context word $c_i$. $\vec{c_i} = W^{(c)}c_i$ which also represents the column $i$ in $W^{(c)}$. The coefficients of $W^{(w)}$ and $W^{(c)}$ will be called $\theta$.

Consider a word-context pair $(w, c)$. Does it comes from the data? The idea is to convert similarity value in vector space into probability using the sigmoid ($\lim_{x\to<-5}\sigma(x) = 0$ ; $\sigma(0) = \frac{1}{2}$ ; $\lim_{x\to>5}\sigma(x) = 1$).

$$P(D = 1|w, c, \theta) = \frac{1}{1 + e^{-\vec{c}\cdot\vec{w}}} = \sigma(\vec{c}\cdot\vec{w})$$

If we want to use a brute force method, then the optimization problem can be formulated by:

- Assume positive examples $D$ for $(w, c)$ and negative examples $\widetilde{D}$, not occurring in the document collection.

- Find $\theta$ such that:

$$\begin{aligned}\theta &= \underset{\theta}{\text{argmax}} \prod_{(w,c)\in D} P(D = 1|w, c, \theta) \cdot \prod_{(w,c)\in\widetilde{D}} P(D = 0|w, c, \theta) \\ &= \underset{\theta}{\text{argmax}} \sum_{(w,c)\in D} log(\sigma(\vec{c}\cdot\vec{w})) + \sum_{(w,c)\in\widetilde{D}} log(\sigma(-\vec{c}\cdot\vec{w}))\end{aligned}$$

Then the **loss function** $J$ (to minimize) can be defined over all terms in the vocabulary by:

$$J(\theta) = \frac{1}{s}\sum_{t=1}^{s} J_t(\theta)$$

$$J_t(\theta) = -log(\sigma(\vec{c}\cdot\vec{w_t})) - \sum_{k=1}^{K} log(\sigma(-\vec{c_k}\cdot\vec{w_t}))$$

where $\vec{c_k}$ are negative examples of context words takes from a set $P_n(w_t)$. How to obtain these negative examples?

The negatives samples are taken from $P_n(w) = V \setminus C(w)$. An empirical approach consists of: If $p_w$ is the probability of word $w$ in collection, choose the word with probability $p_w^{3/4}$. Then less frequent words are sampled more often. Practically we approximate the probability by sampling a few non-context words.

Given the loss function, we can use a **stochastic gradient descent** to approximate the minimal loss function. For every $(w_t, c) \in D$, update $\theta$ with

$$\theta^{new} = \theta^{old} - \alpha\nabla_\theta J_t(\theta^{old})$$

For $J_t$ updates only affect rows in $W^{(w)}$ and $W^{(c)}$ that correspond to $\vec{w_t}$, $\vec{c}$ and $\vec{c_k}$. We update a single column, one by one.

$$\vec{w_t^{new}} = \vec{w_t^{old}} - \alpha\frac{\partial}{\partial\vec{w_t}} J_t(\vec{w_t^{old}}, \vec{c^{old}}, \vec{c_1^{old}}, \ldots, \vec{c_K^{old}})$$

where

$$\frac{\partial J_t}{\partial\vec{w}} = -(1-\sigma(\vec{c}\cdot\vec{w}))\vec{c} + \sum_{k=1}^{K}(1-\sigma(-\vec{c_k}\cdot\vec{w}))\vec{c_k}$$

When stochastic gradient descent is used, only rows that contain the word or some context word in the loss function need to be updated. This implies that the data has to be organized that these rows can be efficiently accessed (e.g. using hashing).

Finally we use $W = W^{(w)} + W^{(c)}$ as the low-dimensional representation. We obtain the following properties for word embeddings:

- Similar terms are clustered
- syntactic and semantic relationships encoded as linear mappings
- Dimensions can capture meaning

It is possible to see syntactic and semantic relationships using word embeddings.(see Figure 19 and Figure 20)

Use of word embedding models:

- Document search: use word embedding vectors as document representation.
- Thesaurus construction and taxonomy induction: search engine for semantically analogous/related terms
- Document classification: use word embedding vectors of documents terms as features

There exists alternative approaches (e.g. CBOW, GLOVE) that we won't see here (lecture 5 slide 16).

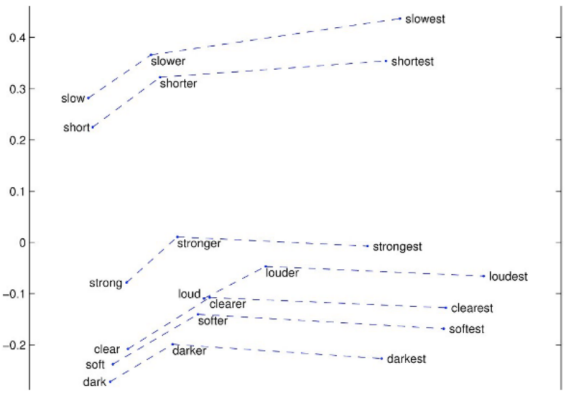


Figure 19: Syntactic relationships

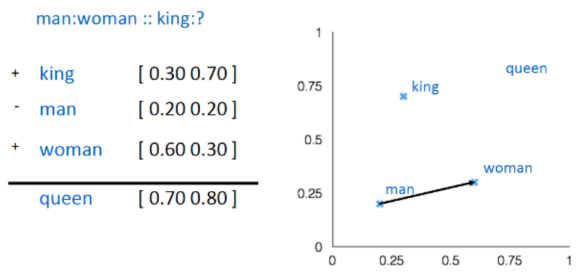


Figure 20: Semantic relationships

### 1.4.2 Link-based ranking

Firstly see some definitions:

1. **Anchor text** describes content of referred document (e.g. "This is a link to Google." → anchor text: "This is a link to Google.").
2. **Hyperlink** is a quality signal.

Anchor text might contain useful information (e.g. blacklist.org pointing to fake epfl website with anchor text: "attention: spam"). We can score anchor text with a weight depending on the authority of the anchor page's website (e.g. if we assume that content from cnn.com is authoritative, then trust (more) its anchor text). To avoid self-promotion we give lower weights to links within the same site.

Indexing anchor text might be useful to locate spammers. Malicious users could create spam pages that point to web pages and try to relate it to contents that serve their interests (e.g., higher the quality of preferred pages by adding links, lower the quality of the undesired page by attaching negative anchor text). That this is happening can be seen from analyzing the in-degree distribution of Web pages. The Figure 21 shows a standard log-log representation of the in-degree vs. the frequency of pages. Normally this relationship should follow a power-law, which shows in a log-log representation as a linear dependency. In real Web data, we see that this power law is violated, and that certain levels of in-degrees are over-represented. This can be attributed to link spamming, which does create moderate numbers of additional links on Web pages.

### 1.4.3 Link-based ranking: PageRank

Simplest idea: more incoming links, higher the relevance. But you can join a link farm (i.e. a group of websites that heavily link to one another). Thus simple link counting is not appropriate.

Basic idea: Imagine a user doing a **random walk** on web pages (i.e. start at a random page. At each step, leave the current page along one of the links on that page, with *same probability*). Use the number of visits in a long run for a given website as its score (i.e. more visits, higher the score).

**Random walker**:

$$P(p_i) = \sum_{p_j : p_j \to p_i} \frac{P(p_j)}{C(p_j)}$$

where $N$ =#web pages, $C(p)$ =#outgoing links of page $p$, $P(p_i)$ = **relevance** = probability to visit page $p_i$, where $p_i$ is pointed by pages $p_1$ to $p_N$. This model takes into account the #of referrals and the relevance of referrals.

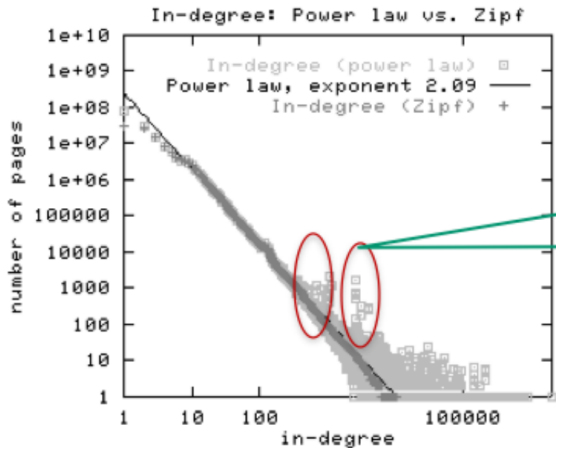


Figure 21: Green: spammers violating power laws!

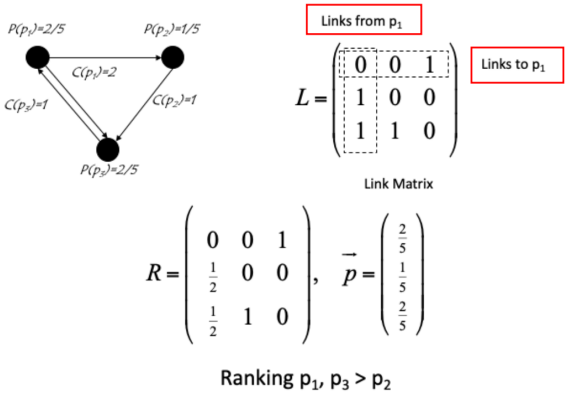


Figure 22: PageRank example

The transition matrix for a random walker: $P(p_i)$ can be reformulated as

$$R_{ij} = \begin{cases} \frac{1}{C(p_j)} & \text{, if } p_j \to p_i \\ 0 & \text{, otherwise} \end{cases} \tag{1}$$

$$\vec{p} = (P(p_1), ..., P(p_n))$$
$$\vec{p} = R \cdot \vec{p}$$
$$||\vec{p}|| = \sum_{i=1}^{n} p_i = 1$$

The vector of page relevance values is the eigenvector with the largest eigenvalue of the matrix $R$. See implementation in the serie "04.Relevance_Feedback".

```
eigenvalues, eigenvectors = np.linalg.eig(R)
p = eigenvectors[:,0] / np.linalg.norm(eigenvectors[:,0], 1)
```

However the eigenvalues method is not scalable. We could use the iterative method.

*Example: see Figure 22. But if we modified this example and delete the link $p_3 \to p_1$, then $\vec{p} = (0, 0, 0)$ which implies no ranking. We need something against this problem.*

Adding **teleportation**: To avoid dead-ends and pages with no incoming links.

- At a dead end, jump to a random web page
- At any non-dead end, jump to a random web page with some small probability $(1 - q)$.

$$\vec{p} = c(qR \cdot \vec{p} + \frac{1-q}{N}\vec{e}), \text{ where } \vec{e} = (1, \ldots, 1)$$

*Example: see Figure 23*

**Practical computation of PageRank**: Iterative computation

$$\vec{p_0} \leftarrow \vec{s}$$
$$\text{while } \delta > \epsilon :$$
$$\vec{p_{i+1}} \leftarrow qR \cdot \vec{p_i}$$
$$\vec{p_{i+1}} \leftarrow \vec{p_{i+1}} + \frac{1-q}{N}\vec{e}$$
$$\delta \leftarrow ||\vec{p_{i+1}} - \vec{p_i}||$$

where $\epsilon$ is the termination criterion and $s$ is an arbitrary start vector (e.g. $s = e$)

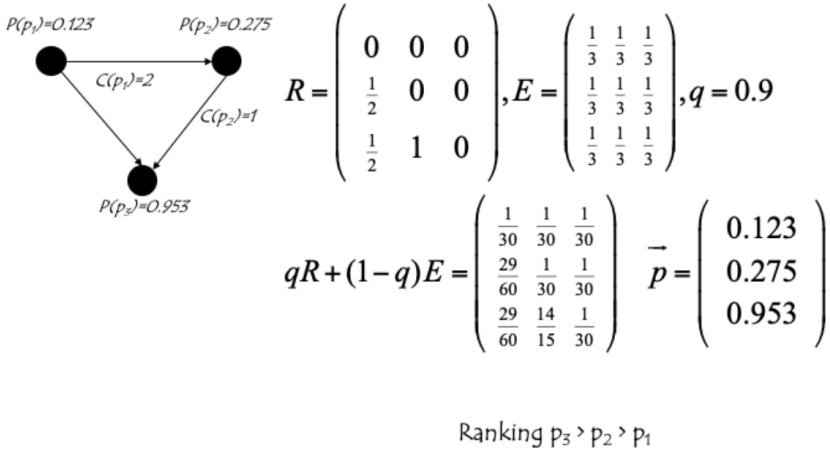


Figure 23: Modified example of PageRank with teleportation

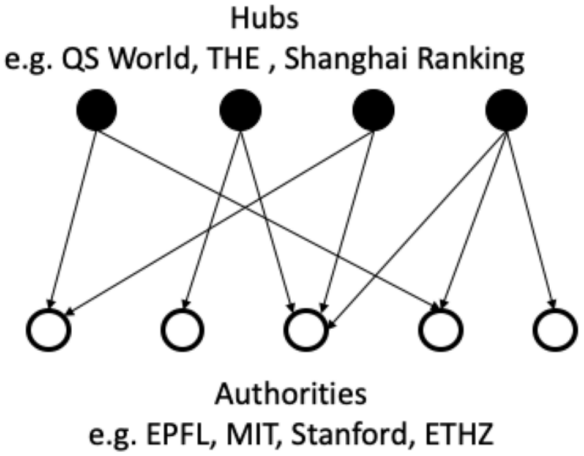


Figure 24: Hubs ad authorities example

### 1.4.4 Link-based ranking: HITS

Key idea of Hyperlink-Induced topic search (HITS): in response to a query, instead of an ordered list of pages, find *two* sets of inter-related pages.

- **Hub pages** are good list of links on a subject (e.g. world top school). That represents the pages that point to many/relevant authorities.
- **Authoritative** pages are referred recurrently on good hubs on the subject (e.g. EPFL). Authorities are pages that are pointed to by many/relevant hubs.

See Figure 24 for an example. Hub-authority ranking identifies not only pages that have a high authority, as measured by the number of incoming links, but also pages that have a substantial "referential" value, having many outgoing links (to pages of high importance).

**Computing hubs and authorities**: Repeat the following updates, for all $p$

$$H(p_i) = \sum_{p_j \in N : p_i \to p_j} A(p_j)$$
$$A(p_i) = \sum_{p_j \in N : p_j \to p_i} H(p_j)$$
$$\sum_{p_j \in N} H(p_j)^2 = 1$$
$$\sum_{p_j \in N} A(p_j)^2 = 1$$

**HITS Algorithm**: In practice, 5 iterations is sufficient to converge! Firstly define $n := |N|$ and $(a_0, h_0) := \frac{1}{n^2}((1, \ldots, 1), (1, \ldots, 1))$

while $l < k$ :

$$l = l + 1$$
$$a_l = (\sum_{p_i \to p_1} h_{l-1,i}, \ldots, \sum_{p_i \to p_n} h_{l-1,i}) \implies a = L^T h$$
$$h_l = (\sum_{p_1 \to p_i} a_{l,i}, \ldots, \sum_{p_n \to p_i} a_{l,i}) \implies h = L \cdot a$$
$$(a_l, h_l) = (\frac{a_l}{|a_l|}, \frac{h_l}{|h_l|}) (\to \text{use np.lingalg.norm}(\ldots, 2))$$

***The normalization at each step is really important!***. We could also use some error comparison (deltas). See implementation in the serie 04.

Assume a n x n link matrix $L$ (if 1 is connected to 2,3,4, then the first line of $L$ is $(0, 1, 1, 1)$. It's the **transpose of the pagerank method for** $L$), $h$ the vectors representing the hubs and $a$ the vector representing the authorities. Then $h = L \cdot a \implies a = L^T h$. It implies that $a^*$ **is the principal eigenvector of** $L^T L$ and $h^*$ **is the principal eigenvector of** $LL^T$ (eigenvalues, eigenvectors = np.linalg.eig(np.matmul(np.transpose(A), A)) is useful).

*Example: Compute the authority values of a graph* $1 \to i$ *with* $i \in 2, 3, 4$. *The result is* $(0, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}})$.

**Base set**: Given a query, obtain all pages mentioning the query (**root set**) and add page that either points to a page in the root set, or is pointed to by a page in the root set. This is the *base set*.

Potential issues with HITS:

- *Topic drift*: off-topic pages can cause off-topic "authorities" to be returned
- *Mutually reinforcing affiliates*: clusters of affiliated pages/sites can boost each others' scores

In terms of implementation, link-based ranking algorithms require the capability to efficiently retrieve the link graph for the Web. This problem is addressed by so-called connectivity servers.

### 1.4.5 Link indexing

**Connectivity server**:

- Support for fast queries on the web graph
  - Which URLs point to a given URL?
  - Which URLs does a given URL point to?

- Stores mappings in memory from URL to outlinks and from URL to inlinks
- Applications
  - Link analysis (PageRank, HITS)
  - Web crawl control
  - Web graph analysis (connectivity, crawl optimization)

**Adjacency lists**: The set of URLs a node is pointing to (or pointed to) sorted in *lexicographical* order. They have two properties:

- **Locality** (within lists): Most links contained in a page have a navigational nature thus their indices are close in lexicographical order. (e.g. epfl.ch ← futuretudiant.epfl.ch/en, futuretudiant.epfl.ch/mobility)
- **Similarity** (between lists): Either two lists have almost nothing in common, or they share large segments of their successor lists. Pages that occur close to each other in lexicographical order tend to have similar lists. (e.g. futuretudiant.epfl.ch/en and futuretudiant.epfl.ch/mobility)

We can exploit locality or similarity to compress the adjacency lists.

*Exploit Locality*: use gap encoding (as in inverted files): $S(x) = (s_1, \ldots, s_k) = (s_1 - x, s_2 - s_1 - 1, \ldots, s_k - s_{k-1} - 1)$. (e.g. *node=16, outdegree=10, successors=[15,16,17,22,23,24,315,316,317,3041]* → *node=16, outdegree=10, successors=[1,0,0,4,0,0,290,0,0,2723]*).

Check that lec5 s61. Not sure for $s_1 - x$

*Exploiting similarity*: Copy data from similar lists.

- Reference list: reference to another list (searched in a neighboring window of nodes)
- Copy list: indicaties which node is copied from reference list
- extra nodes: additional nodes not in reference list

(e.g. *node=16, outdegree=10, successors=[15,16,17,22,23,24,315,316,317,3041]* → *node=16, outdegree=10, successors=[1,0,0,4,0,0,290,0,0,2723]* → *node=16, outdegree=10, ref=1, copy list=[01110011010], extra nodes=[22,316,317,3041]*). See lecture 5 slides 61-62.

**Note**: Exploiting locality with gap encoding may increase the size of an adjacency list. Exploiting similarity with reference lists may increase the size of an adjacency list.

# 2 Data Mining

## 2.1 LEC06 : Frequent Itemsets

### 2.1.1 Introduction to data mining

Data mining is the opposite way of retrieval. Given data you have to find a model that matches the data. The challenge is: we produce an enormous amount of data but most is useless to transform into an human understandable and useful intelligence. The classical example of data mining problem is "market basket analysis". A well-known example was the discovery that people who buy diapers also frequently buy beers. Therefore nowadays one finds frequently beer close to diapers in supermarkets. This type of problem was the starting point of the association rule mining (see an overview in Table 4).

There exists different classes of data mining problems.

- Local properties: Patterns that apply to part of the data (e.g. buy diapers → buy beers)
- Global model:

| Local properties | | Global model | |
|---|---|---|---|
| **Patterns** | | **Descriptive model** | **Predictive model** |
| Association rules | Pattern mining | Clustering | Classification |
| | | Information retrieval | Regression |
| **Explanatory data analysis** | | | |
| Interactive tools | | Visualisation | |
| *Unsupervised* | | | *Supervised* |

Table 4: Data mining overview

- Descriptive **structure** of the data (e.g. 3 types of customers behaviour)
- Predictive **function** of the data: (e.g. if dist(beer, diapers)=1 $\rightarrow$ +10% beer sales)

Each data mining algorithm can be characterized by four aspects:

1. The model representation or patterns structure. (i.e. What we look for?)
2. The scoring functions (i.e. How well the model fits the data set?). This is comparable to the similarity functions used in information retrieval.
3. Optimisation and search (i.e. How to tune the parameters of the model? [opt] ; How to find data satisfying a pattern? [search])
4. Data management. The scalability implementation of the method for large data sets. This corresponds to the use of inverted files in information retrieval.

Keep in mind that data mining (DM) $\neq$ machine learning (ML).

$=$ In DM for data analytics frequently typical ML methods are used. (though not always (e.g. visual mining))

$\neq$ Data: DM is always applied to large datasets, ML not

$\neq$ Scope: DM comprises of the whole process of data integration, cleaning analysis.

$\neq$ Goal: DM aims at detecting unsuspected patterns, ML may have other goals (e.g. winning a game)

### 2.1.2 Association rule mining

**Pattern structure**: We search association rules of the form

$$\text{Body} \rightarrow \text{Head [support, confidence]}$$

where body and head are of the form predicate($x, x \in$ {items}). Body being a property of $x$ and head being a property likely to be implied by the body. The body might be single- or multi-dimensional rules. However, we can transform a multi- to single-dimensional rules using predicate/value pairs as items.
*Example of multi-dimensional-rules:* age(x, {19-25}) & buy(x, {chips}) $\rightarrow$ buy(x, {coke}) [10%, 75%]
*Example:* customer(x, {age=19-25}) & customer(x, {buy=chips}) $\rightarrow$ customer(x, {buy=coke}) [10%, 75%]

*Example: see Table 5*

**Definition of association rules**:

- Set of all items $I$, sub set of $I$ is called **itemset**

| Transaction ID | Items bought |
|---|---|
| 2000 | beer, diaper, milk |
| 1000 | beer diaper |
| 4000 | beer milk |
| 5000 | milk, eggs, apple |

| |
|---|
| p({beer, diaper})=2/4<br>p({diaper}\|{beer})=2/3<br><br>{beer}→{diaper}[50%, 66%] |

| |
|---|
| p({beer, diaper})=2/4<br>p({beer}\|{diaper})=2/2<br><br>{diaper}→{beer}[50%, 100%] |

Table 5: Example support and confidence

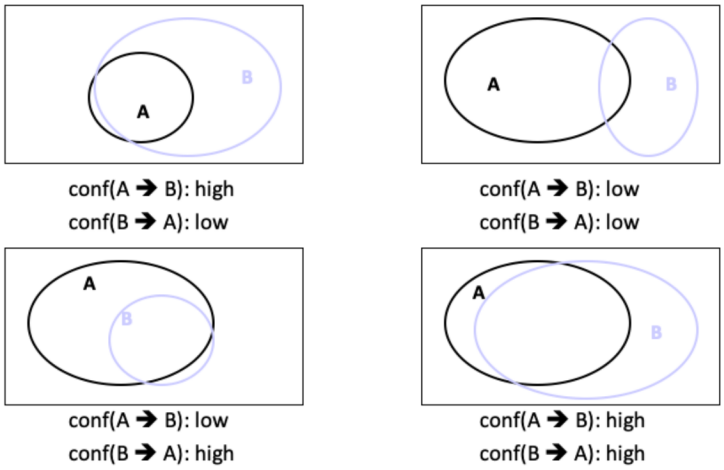


Figure 25: Importance of "direction". Assume p({A,B}) above threshold (high support)

- **Transaction** $(tid, T)$, $T \subseteq I$ itemset, transaction identifier $tid$
- Set of all transactions $D$ (**database**), transaction $T \in D$
- Association rules: $A \rightarrow B[s, c]$
  - $A$, $B$ itemsets $(A, B \subseteq I)$
  - $A \cap B$ empty
  - **Support**: $s = p(A \cup B) = \frac{count(A \cup B)}{|D|}$. Probability that body and head occur in transaction (i.e. p({body, head})). If this number is too small, probably the rule is not relevant.
  - **Confidence**: $c = p(B|A) = \frac{s(A \cup B)}{s(A)}$. Probability that if body occurs, also head occurs (i.e. p({head}|{body})). This indicates to which degree the rule is satisfied, where it is applicable.

The problem of the association rule mining is: Given a database $D$ of transactions $(tid, T)$, find all rules $A \rightarrow B[s, c]$ such that $s > s_{min}$ (high support) and $c > c_{min}$ (high confidence).

**Frequent itemsets using Apriori property/algorithm**:

1. $A \rightarrow B$ can be an association rule, only if $A \cup B$ is a frequent itemset.
2. Any subset of frequent itemset is also a frequent itemset (**Apriori property**): $p(J) > s_{min} \implies p(J' \subseteq J) > s_{min}$
3. Find frequent itemsets with increasing cardinality, from 1 to k, to reduce the search space.

The union of two (k-1)-itemsets that differs only by one item is a **candidate** frequent k-itemset. This step is called the **join step** and is used to construct *POTENTIAL* frequent k-itemsets. This condition is *necessary but not sufficient* to obtain a frequent k-itemset. We need the **prune step** after the join step. A k-itemset in the candidate set $C_k$ (after one join step) might still contain

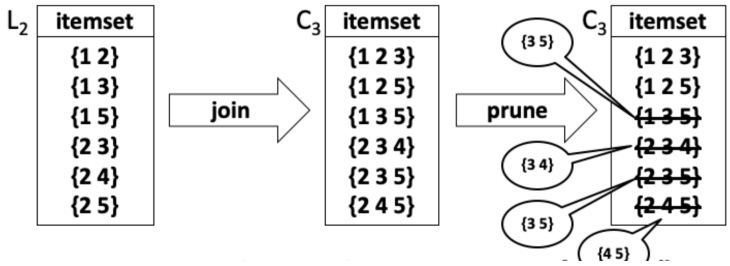


Figure 26: Join-prune example

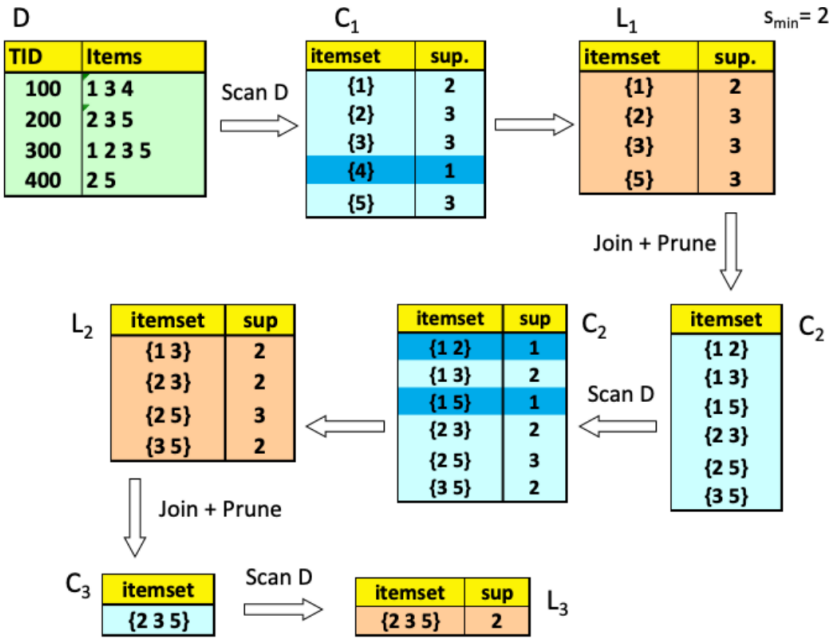


Figure 27: Apriori example

(k-1)-itemsets that are not frequent (k-1)-itemsets. Eliminate them (that is the prune step). The reader can see the Apriori algorithm in Figure 28. Note that after the join step, the number of (k+1)-itemsets can be equal, lower or higher than the number of frequent k-itemsets. Once the frequent itemsets are found with Apriori, the derivation of pertinent rules is straightforward (see Figure 29).

*Example: see Figure 26 and Figure 27*

**Alternative measure of interest**:

- **Added value**: $AV(A \to B) = confidence(A \to B) - support(B)$. Interesting rules are those with high positive or negative interest value (usually $> 0.5$).
- **Lift**: $Lift(A \to B) = \frac{confidence(A \to B)}{support(B)}$

For quantitative attributes the search for association rule is more complex. A simple approach is to statically or dynamically discretize quantitative attributes into categorical attributes.

**Improving Apriori for large datasets**:

- **Transaction reduction**: A transaction that does not contains any frequent k-itemsets is useless in subsequent scans
- **Sampling**: Mining on a sampled subset of database $D$, with a lower support.
  1. Randomly sample transactions with probability $p$. if we know $m$ transactions are randomly sorted, we can choose the first $m \cdot p$ ones.

```
k := 1, L_k := { J ⊆ I : |J|=1 ∧ p(J) > s_min }          //frequent items
while L_k != ∅
        C'_{k+1} := JOIN(L_k)              // join itemsets in L_k that differ by one element
        C_{k+1} := PRUNE(C'_{k+1})                  // remove non-frequent itemsets
        for all (id,T) ∈ D                          // compute the frequency of candidate
                for all J ∈ C_{k+1} , J ⊆ T
                        count(J)++
                end for
        end for
        L_{k+1} := { J ⊆ C_{k+1} : p(J) > s_min }   // add candidate frequent itemsets
        k := k+1
end while
return ∪_k L_k
```

Figure 28: Apriori algorithm

```
R := ∅                                              // initial set of rules
L := Apriori(D)                                     // set of frequent itemsets
for all J ∈ L
        for all A ⊆ J, A != ∅
                r := A ➔ J \ A                      // create candidate rule
                if c(A ➔ J \ A) = s(J)/s(A)> c_min
                        R := R ∪ r
                end if
        end for
end for
```

Figure 29: Select the pertinent rules

| TID | Items |
|---|---|
| 1 | {a,b} |
| 2 | {b,c,d} |
| 3 | {a,c,d,e} |
| 4 | {a,d,e} |
| 5 | {a,b,c} |
| 6 | {a,b,c,d} |
| 7 | {a} |
| 8 | {a,b,c} |
| 9 | {a,b,d} |
| 10 | {b,c,e} |

Figure 30: Transaction data set for FP-Tree (step 1)

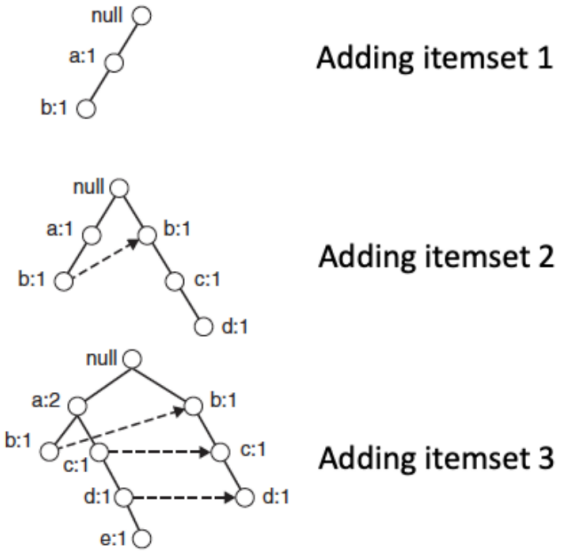


Figure 31: Example FP-Tree step 2

2. Detect frequent itemsets with support $p \cdot s$
3. Eliminate false positive by counting frequent itemsets on complete data after discovery. They can be reduced by choosing lower threshold (e.g. $90\% \cdot p \cdot s$)

- **Partitioning** (SON algorithm): Any itemsets that is potentially frequent in $D$ must be frequent in at least one of the partitions of $D$.

  1. Divide transactions in $\frac{1}{p}$ partitions and repeatedly read partitions into main memory.
  2. Detect in-memory algorithm to find all frequent itemsets with support threshold $p \cdot s$
  3. An itemsets becomes a candidate if it is found to be frequent in at least one partition.
  4. On a second pass, count all the candidate itemsets and determine which are frequent in the entire set of transactions.

  It can also be implemented in a MapReduce idea.

**FP Growth**: We will now see an other approach to the frequent itemsets that works without candidate itemsets generation. It is working in two steps:

1. Build a data structure, called the FP-tree (requires 2 passes over the dataset)
2. Extract frequent itemsets directly from the FP-tree.

**FP-tree construction**: requires 2 passes over the dataset (see Figure 30, Figure 31 and Figure 32)

1. Compute the support for each item. Pass over the transaction set and sort items in order of decreasing support.
2. Construct the FP-tree data structure. Tree is expanded one itemsets at a time.

**FP-tree frequent itemsets extraction**:

1. *Bottom-up* approach: for each item extract the tree with paths ending in the item.
2. *Divide and conquer*: Start whether the item has sufficient support (e.g. 2).

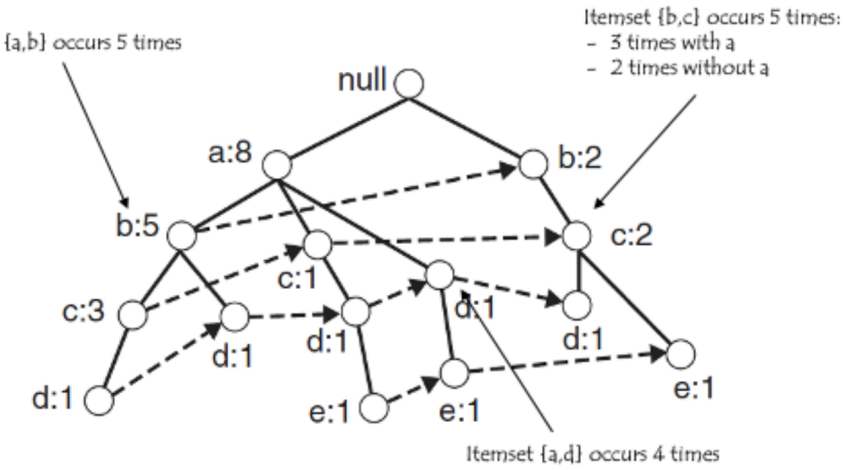


Figure 32: FP-Tree resulting after step 2

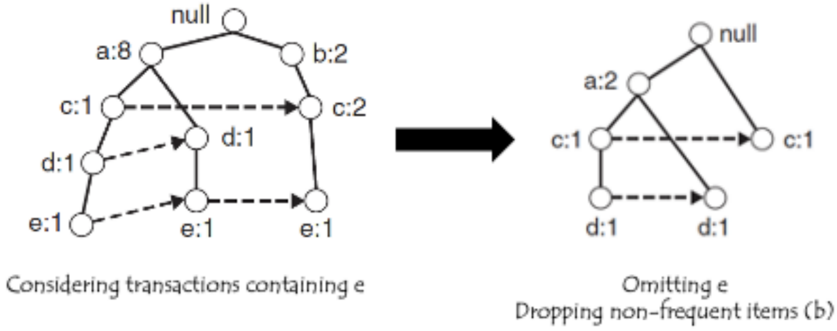


Figure 33: Conditional FP-Tree

- Add up counts along the list of the item (e.g. item $e$ has support 3) (see Figure 32 where we selected $e$ and applied bottom-up).
- If sufficient support, check for itemsets ending in the item (e.g. Assume $e$ is frequent, check whether $de$, $ce$, $be$ and $ae$ are frequent in order of lowest support!). For that compute the conditional FP-Tree

**Conditional FP-Tree**: Removing a chosen itemset from the tree and dropping items that are not frequent. It can be derived from the extracted tree (see Figure 33).

1. Update support counts to itemsets containing the item
2. Remove the nodes of the item
3. Remove nodes with insufficient support count.

From the conditional FP-Tree for $e$ we are able to find the support for $de$, $ce$, $be$, $ae$ ($s(de) = s(ce) = s(ae) = 2$ and $s(be) = 0$). We also can compute the conditional FP-Tree for $de$ and we find that $ade$ has a support count of 2.

Finally the result can be seen in Table 6

**FP Growth advantages and disadvantages**:

\+ Only 2 passes over the dataset

\+ Compress dataset

\+ (Generally) much faster than Apriori

| Suffix | Frequent Itemsets |
|---|---|
| e | {e}, {d,e}, {a,d,e}, {c,e}, {a,e} |
| d | {d}, {c,d}, {b,c,d}, {a,c,d}, {b,d}, {a,b,d}, {a,d} |
| c | {c}, {b,c}, {a,b,c}, {a,c} |
| b | {b}, {a,b} |
| a | {a} |

Table 6: Result after FP-Tree approach

- Works less efficiently for high support threshold
- Has to run in main memory
- Difficult to find distributed implementation

To summarize the components of data analytics we have:

1. Pattern structure/model representation: association rule
2. Scoring function: support, confidence
3. Optimisation and search: Join, prune, FP-tree, ordering of items
4. data management: transaction reduction, partitioning, sampling

## 2.2 LEC07 : Clustering

Given a datasets of objects described by attributes we want to build a model that assigns objects to a class/label. There exists two approaches:

- **Descriptive (clustering)**: We only the data items as indicated by points in a n-dimensional space.
- **Predictive (classification)**: We already know the classification of the data (i.e. that object belongs to that class). The classification method infers conditions on the properties of the data objects. That allows to predict the membership to a specific class.

We will see classification in the next lecture. Let's begin with clustering.

**Clustering** belongs to unsupervised machine learning. The idea is to partition a set of objects into clusters (i.e. similar objects belongs to the same cluster). If the reader wants to know how to use clusters for information retrieval, take a look at the text of lecture 7 slide 4. The algorithm does not label the clusters but the user could do that by inspecting the results.

The *problem formalization* for clustering is:

- *Problem*: Given a database $D$ with $n$ data items described by $m$ attributes
- *Find*: partition of $D$ into $k$ clusters
- *such that*: Intra-cluster similarity is high and inter-cluster similarity is low.

*Characteristics* of clustering methods:

- Quantitative:
  - Scalability: high $n$
  - Dimensionality: high $m$

In general high-dimensional data (large $m$) is too sparsely distributed to identify meaningful clusters. Then we project the data to a lower dimensional space. It makes the clustering sensitive of the choice of the dimensions used for projection.

- Qualitative:
  - Different types of attributes (e.g. numerical, categorical)
  - Different types of shapes (e.g. spheres, hyperplanes)

  It concerns the ability of dealing with continuous as well as categorical attributes, and the type of clusters that can be found. Many clustering methods can detect only very simple geometrical shapes (e.g. spheres, hyperplanes)

- Robustness:
  - Sensitivity to noise and outliers
  - Sensitivity to the processing order

- User interaction:
  - Incorporation of user constraints (e.g. number of clusters, maximal size of clusters)
  - Interpretability and usability

There exists different methods for clustering. We will begin with partitioning methods (i.e. **K-means** and its friends) then we will see an advanced clustering method called **DBSCAN**. Finally we will discuss about how to evaluate clustering quality.

**Partitioning methods**:

- *Problem*: Given a database $D$ of $n$ objects, split $D$ into $k$ sets $C_1, \ldots, C_k$ such that $C_i \cap C_j = \emptyset$ for all $C_i \neq C_j$ and $\bigcup_i C_i = D$
- The optimal algorithm is to enumerate all partitions and pick the best (impossible in practice)
- heuristic algorithms: (points are not part of the dataset ; objects are)
  - **K-means**: cluster is represented by the **point** whose mean distance with the objects in the cluster is minimal
  - **K-medoids**: cluster is represented by the **object** whose mean distance with the objects in the cluster is minimal
  - **K-medians**: cluster is represented by the **point** whose *median* distance with *all* the objects in the cluster is minimal

  *Note: To compare pictures of faces, k-mediods seems more appropriate.*

**K-means algorithm**: This is an iterative algorithm. See Figure 34 for an example.

- Algorithm:
  1. Initialize $k$ random points as cluster centers
  2. assign each object to the "nearest" cluster center. It generates a partitioning $C_1, \ldots, C_k$ of $D$
  3. While partitioning changes (i.e. till it converges):
     - for each cluster, calculate the centroid of the points and set it as new cluster center

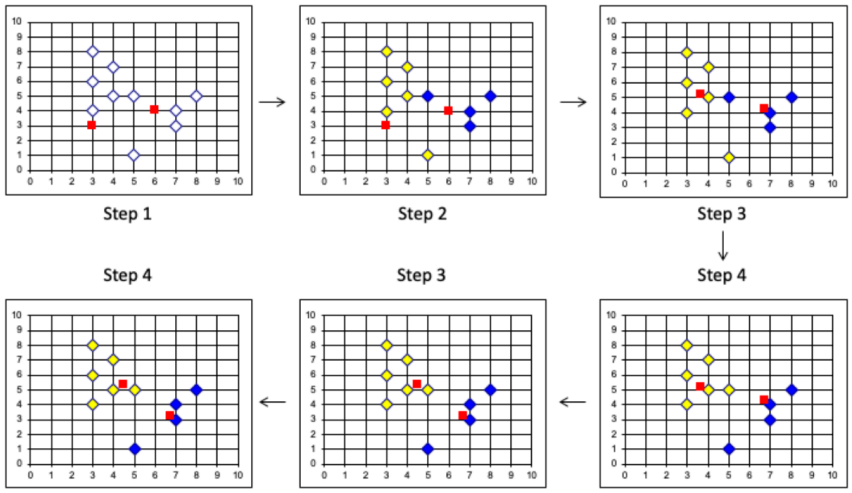


Figure 34: K-means example

– assign each point to the "nearest" cluster center

*Score function* of K-means: Find $C_i$ that minimizes loss function $J$

$$J = \frac{1}{n} \sum_{i=1}^{k} \sum_{x_j \in C_j} \|x_j - \mu_i\|^2 \text{ , where } \mu_i = \frac{1}{|C_i|} \sum_{x_j \in C_i} x_j$$

$\mu_i$ = centroid of $C_i$

- Advantages and disadvantages:
  - \+ Efficient: $O(tkn)$
  - \+ $n$ = #objects, $k$ = #clusters, $t$ = #iterations ; ($k$, $t << n$)
  - \+ Converges fast
  - \- Often terminates at a local minimum
  - \- Need a distance functions to computes the mean (**matching coefficient** for k-means: $a$ and $b$ are objects with $m$ attributes)

$$d(a, b) = \frac{|\{i|a_i \neq b_i\}|}{m}$$

  - \- Need to **specify** $k$ **in advance** (**model parameter**. To find a suitable number of k we rely on the following intuition. The higher the value of k, the lower the within-cluster sum of squares. However, creating too many clusters is useless. Thus we want to find the smallest k after which the within-cluster sum of squares decreases "slowly" (elbow method))
    1. Execute for $k = 1, 2, 3, 4, \ldots$
    2. Plot the score $J$ against $k$
    3. Pick $k$ such that $J(k) \sim J(k + 1)$
  - \- Does not handle noisy data and outliers
  - \- Clusters only have convex shapes

Let's talk a bit about **advanced clustering** techniques. Distance measures for mixed attributes.

**Density-based clustering** (DBSCAN): Clustering based on a local, density-based criterion. Properties:

- Discovers clusters of arbitrary shape
- Handles noise
- Clusters in one scan
- No predetermined number of clusters
- Model parameters: density parameters

Basics notions/definitions (see lecture 7 slide 21-25 for drawing):

- **Core point**:
  - A distance metric $d$ is given.
  - $\epsilon$**-neighborhood**: $N_\epsilon(q) = \{p | d(p,q) < \epsilon\}$
  - A point $q$ is a *core point* if $|N_\epsilon(q)| \geq \mu$
  - $\epsilon$ and $\mu$ are model parameters
- **Border point**:
  - A point $p$ is **directly density-reachable** from $q$ if $p \in N_\epsilon(q)$ and $|N_\epsilon(q)| \geq \mu$
  - A point that is directly density-reachable but not a core point is a *border point.*
  - A point that is not directly density reachable is an **outlier**
- Direct density-reachability induces a **directed graph** on the points
- **Density-reachable** (not direct ; asymmetric relationship): A point $p$ is *density-reachable* from a point $q$ with $\epsilon, \mu$ if there is a chain of points $p_1, \ldots, p_n$ with $p_1 = q$ and $p_n = p$ such that $p_{i+1}$ is directly density-reachable from $p_i$.
- **Density-connected** (symmetric relationship): A point $p$ is *density-connected* to a point $q$ with $\epsilon, \mu$ if there is a point $r$ such that both, $p$ and $q$ are density-reachable from $r$ with $\epsilon, \mu$
- **Cluster**: A cluster $C$ satisfies:
  - **Maximality**: if $q$ in $C$ is a core point, and $p$ is density reachable from $q$, then also $p$ in C
  - **Connectivity**: any two points in $C$ must be density connected $\implies$ a cluster contains at least one core point.
  - The set of clusters is unique
  - Clusters are not necessarily disjoint

**DBSCAN Algorithm** (see lecture 7 slide 28-29):

1. Initialization ($O(n^2)$): Construct a directed graph $G$ using direct density-reachability. Initialize
   - $V_{core}$ = set of core points
   - $P$ = set of all points
   - set of clusters $C = \{\}$
2. Cluster construction ($O(n^2)$): while $V_{core}$ not empty (once $V_{core}$ is empty, we know that no more new clusters can be found):

(a) Select a point $p$ from $V_{core}$ and construct $S(p)$, the set of all points density-reachable from $p$: BFS on $G$ starting from $p$ (pseudocode for BFS in scripts folder)

(b) $C = C \cup \{S(p)\}$

(c) $P = P \setminus S(p)$

(d) $V_{core} = V_{core} \setminus S_{core}(p)$ where $S_{core}(p) =$ core points in $S(p)$

Mark remaining points in $P$ as unclustered.

*Note: In density-based clustering only the border points can belongs to multiple clusters.*

*Note: When executing DBSCAN the result is independent of the order of choosing initial core points.*

*Note: Even in the case of dimension $d = 3$ the worst case complexity is $\Omega(n^{\frac{4}{3}})$*

**Evaluating clustering quality**: We often have no ground truth available (do not know what is the "correct" clustering). We have to define measures that characterize of how the properties of *high intra-cluster similarity* and *low inter-cluster similarity* are achieved and maybe consider also other properties of clusters. We will see the **average silhouette width** (ASW): For data point $x_i \in C_i$:

- Measure for intra-cluster distances (average of the distance of all the points in the same cluster):
$$a(x_i) = \frac{1}{|C_i| - 1} \sum_{x_j \in C_i, i \neq j} d(x_i, x_j)$$
- Measure for inter-cluster distances (minimum average of the distance of all the points in an other cluster):
$$b(x_i) = \min_{i \neq k} \frac{1}{|C_k|} \sum_{x_j \neq C_k} d(x_i, x_j)$$
- Silhouette value: For a data point characterizes how the point "fits" into its own cluster. The *average* silhouette value over all data points characterizes the quality of clustering.
$$s(x_i) = \begin{cases} \frac{b(x_i) - a(x_i)}{\max(a(x_i), b(x_i))} & \text{, if } |C_i| > 1 \\ 0 & \text{, if } |C_i| = 1 \end{cases}$$
- Silhouette coefficient: $SC = \max_k \bar{s}(k)$
  1. Assume the clustering method produces a specified number of clusters $k$ (e.g. k-means)
  2. Compute $\bar{s}(k)$ as the average of the silhouette values of all data points for the clustering of size $k$
  3. The silhouette coefficient then characterizes the overall quality of the clustering method.

## 2.3 LEC08 : Classification

Given a dataset of *objects* described by *attributes*, build a model that assigns objects to a class. See Figure 35. We still have the same idea as before for training, test (unseen data) and prediction set (unseen and unclassified data).

**Characteristics of classification methods**:

- Predictive accuracy (75% accuracy means that the model classifies 75% of the test set.)
- Speed and scalability
  - Time to build the model

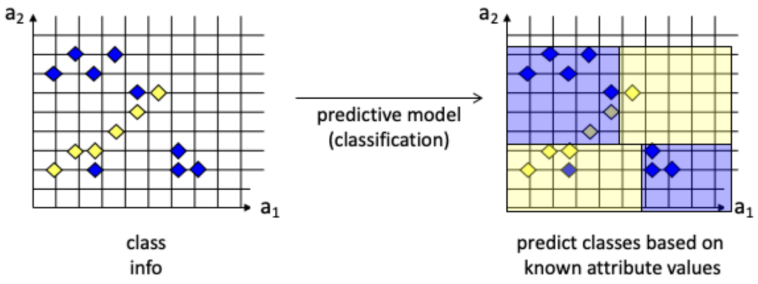


Figure 35: Classification: attributes $(a_1, a_2)$ separated in intervals

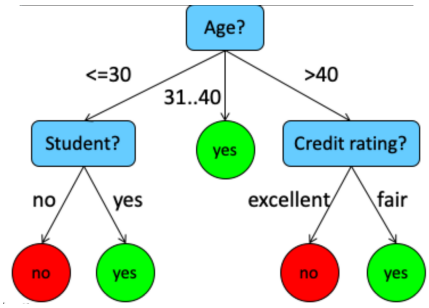


Figure 36: Decision tree (buy a new computer?)

  - Time to use the model
  - In memory vs. on disk processing
- Robustness: Handling noise, outliers and missing values
- Interpretability:
  - Understanding the model and its decisions (black box) vs. white box
  - Compactness of the model

**Decision Trees**: Nodes are tests on a single attribute. Branches are attribute values. Leaves are marked with class labels. See example Figure 36.

**Decision Trees: Algorithm**:

- Tree construction (top-down divide-and-conquer strategy)
  - At the beginning, all training samples belong to the root
  - Examples are partitioned recursively based on a selected "most discriminative" attribute (see attribute selection below)
  - Discriminative power determined based on information gain
- Partitioning stops if
  - All samples belong to the same class → assign the class label to the leaf
  - There are no attributes left → majority voting to assign the class label to the leaf
  - There are no samples left

**Attribute selection**:

- **Definition**: At a given branch in the tree, the set of samples $S$ to be classified has $P$ positive and $N$ negative instances. The entropy of the set $S$ is

$$H(P,N) = -\frac{P}{P+N}\log_2(\frac{P}{P+N}) - \frac{N}{P+N}\log_2(\frac{N}{P+N})$$

  Note
  - If $P = 0$ or $N = 0$, then $H(P,N) = 0 \rightarrow$ no uncertainty
  - If $P = N$, then $H(P,N) = 1 \rightarrow$ maximum uncertainty
  - $H(P,N) = H(N,P)$
  - For example $P = 9$ and $N = 5$ then $H(P,N) = 0.94$ i.e. 0.94 bits are required to decide the class of one instance.

An example can be seen in lecture 8 slide 17.

- **Information gain**: Attribute $A$ partitions $S$ into $S_1, S_2, \ldots, S_v$. Entropy of attribute $A$ is
$$H(A) = \sum_{i=1}^{v} \frac{P_i + N_i}{P + N} H(P_i, N_i)$$
The information gain obtained by splitting $S$ using $A$ is $Gain(A) = H(P, N) - H(A)$. Thus, we split on the argument that produces the higher gain (See example in lecture 8 slide 19).

- **Pruning**: The construction phase does not filter out noise $\implies$ *overfitting*. Let's see some pruning strategies
  - Stop partitioning a node when large majority of samples is positive or negative (i.e. $\frac{N}{P+N}$ or $\frac{P}{P+N} > 1 - \epsilon$).
  - Build the full tree, then replace nodes with leaves labelled with the majority class, if classification accuracy does not change.
  - Apply Minimum Description Length (**MDL**) principle: Let $M_1, \ldots, M_n$ be a list of candidate models (i.e. trees). The best model is the one that minimizes $L(M) + L(D|M)$ where
    - $L(M)$ is the length of the description of the model in bits. (#nodes, #leaves, #arcs, ...)
    - $L(D|M)$ is the length of the description of the data when encoded with the model in bits (#misclassifications)

- **Extraction classification rules from trees**: The goal is to represent the knowledge in the form of IF-THEN rules with the following constraints:
  - One rule is created for each path from the root to a leaf.
  - Each attribute-value pair along a path forms a conjunction.
  - The leaf node holds the class prediction.

**Continuous attributes**: Cannot have separate branch for each value → use **binary decision trees**:

- For continuous attributes A a split is defined by $val(A) < X$
- For categorical attributes A a split is defined by subset $X \subseteq$ domain(A)

To split continuous attributes we use the same idea as before (see Figure 37):

1. Sort the data according to attribute value.
2. Determine the value of X which maximizes information gain by scanning through the data items.
3. Note that a relevant decision point exists only if the class label changes.

**Scalability continuous attribute splits**: The problem is that after each step we have to traverse again the attributes to resort them for each investigation. We need a better approach. *Idea*: Presorting of data and maintaining order throughout tree construction (requires separate sorted attribute tables for each attribute). We have two situation:

- For the table that keeps the attribute that was used in the split, the table needs just to be partitioned into two subtables, maintaining the order.

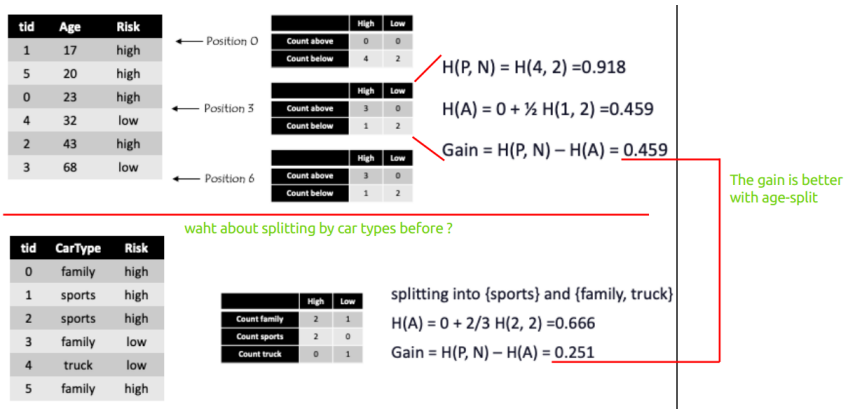


Figure 37: Example of continuous split

- For the other attributes we have to select the subtables corresponding to the instances of the two partitions that have been formed. To that end a temporary *hash table* is constructed that allows to associate to each data item its partition. Then the attribute table is scanned and partitioned using the hash table to decide for each entry to which partition it belongs. Here we keep the order from the original table.

**Characteristics of decision tree induction**:

\+ Automatic feature selection

\+ Minimal data preparation

\+ Non-linear model

\+ Easy to interpret and explain

\- Sensitive to small perturbation in the data

\- Tend to overfit

\- No incremental updates

**Properties of decision tree induction**:

- *Model*: flow-chart like tree structure
- *Score function*: classification accuracy
- *Optimisation*: top-down tree construction + pruning
- *Data management*: avoiding sorting during split

**Ensemble Methods**: The idea is to take a collection of simple or weak learners and to combine their results to make a single, strong learner. There exists multiple types:

- **Bagging**: train learners in parallel on different samples of the data, then combine outputs through voting or averaging
- **Stacking**: combine model outputs using a second stage learner like linear regression
- **Boosting**: train learners on the filtered output of other learners

**Random Forest** (still considered as the most robust method): Learn $K$ different decision trees from independent samples of the data (bagging). Then vote between different learners, so models should not be too similar. *Aggregate output*: majority vote. This methods works even if the individual classifiers are not very good (e.g. 35% errors) their aggregate will be very strong (e.g. 6% errors).

$$P(\text{wrong prediction}) = \sum_{i=\left\lfloor \frac{n}{2} \right\rfloor + 1}^{n} \binom{n}{i} \epsilon^i (1-\epsilon)^{n-i}$$

Where $n$ is the number of base classifiers. See lecture 8 slide 37 for an example. There exists two sampling strategies:

- Sampling data: select a subset of the data → each tree is trained on different data
- Sampling attributes: select a subset of attributes → corresponding nodes in different trees (usually) don't use the same features to split.

**Random Forests Algorithm**:

1. Draw $K$ bootstrap samples of size $n$ from original dataset, with replacement (bootstrapping)
2. While constructing the decision tree, select a random set of $m$ attributes out of the $p$ attributes available to infer split (feature bagging)

Typical parameters: $m \approx \sqrt{d}$, or smaller ; $K \approx 500$ *Note: The computational cost for constructing a RF with K as compared to constructing K trees on the same data is on average smaller.*

**Characteristics of Random Forests**:

+ Ensemble can model extremely complex decision boundaries without overfitting
+ Probably the most popular classifier for dense data ($\leq$ a few thousand features)
+ Easy to implement (train a lot of trees)
+ Parallelizes easily, good match for MapReduce
- Deep Neural networks generally do better
- Needs many passes over the data (at least the max depth of the trees)
- Relatively easy to overfit (hard to balance accuracy vs. fit tradeoff)

## 2.4 LEC09 : Classification Methodology

### 2.4.1 Classification methodology

Data is sometimes questionable, misleading or erroneous. Trustworthiness + expertise = credibility. This is a classification problem.

**Classification pipeline main steps:**

1. Data collection and preparation (domain knowledge and understanding essential)
   (a) Feature identification
   (b) Labelling
   (c) Discretization
   (d) Feature selection

(e) Feature normalization

2. Model training, selection and assessment (understanding potential and limits of machine learning essential)

   (a) Selecting performance metrics

   (b) Model selection

   (c) Organizing training and test data

### 2.4.2 Data collection and preparation

**Feature identification**: The first step is collecting data related to the classification task. This implies the definition of attributes (or features) that describe a data item and the class label. In general domain knowledge is needed. There exists different types of features

- Numerical (e.g. age, temperature)
- Ordinal (e.g. phone code)
- Categorical (e.g. student, weather)

The type of the feature has an impact on the type of classifier that can be used, as some can work only with one type of feature.

**Labelling**: Collecting lot of data is easy. Labelling them is time consuming, expensive and difficult and sometimes impossible. There exists different way to obtain labels:

- Ask experts or do it yourself: expensive, boring, low volume
- Ask the crowd (**crowd-sourcing**): Less expensive, popular, unreliable
- Find some complementary information sources (distant learning)

The problem with crowd-sourcing is that there are different types of crowd-workers (see Figure 38):

- Truthful: Expert, normal
- Untruthful:
  - Sloppy: *many wrong answers* due to limited knowledge or misunderstanding
  - Uniform spammer: *same answer* for any question
  - Random spammer: *random answer* for every question

After each crowd-workers labelled the objects, how to aggregate the result? There are two main classes of **answer aggregation algorithms**:

- **Non-iterative**: It takes the matrix of answers provided by the workers, pre-process it and produces an estimate of the probability of the most likely answer to be correct. See Figure 39.
  - **Majority Decision** (MD): Very sensitive to spammers since every workers have the same weight in voting decision.
    1. No pre-processing step

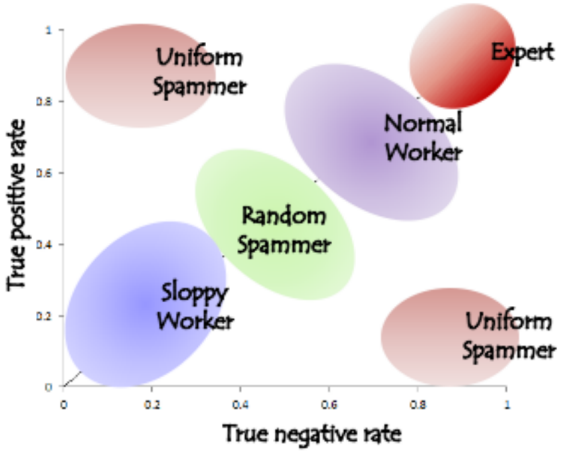


Figure 38: Types of crowd-workers

2. Estimate $P(x_j = l)$ using

$$P(x_j = l) = \frac{1}{N} \sum_{i=1}^{N} (1 | a_i(x_j) = l)$$

where

$$\begin{aligned} x_j &= \text{web page to label} \\ N &= \#\text{workers} \\ l &= \text{label} \\ a_i(x_j) &= \text{answer of worker } i \text{ to web page } x_j \end{aligned}$$

– **Honey Pot** (HP):

1. Pre-processing step:
   (a) Insert web pages $\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_k$ for which the label $\hat{l}_1, \hat{l}_2, \ldots, \hat{l}_k$ are known
   (b) Remove workers and corresponding answers that fail at correctly labelling more than $m\%$ of web pages (spammer or sloppy)
2. Same decision rule as MD.

- **Iterative**: They take the matrix of answers provided by the worker and produce an estimate of the probability that a web page X is labelled by label L. With this estimate they update the expertise of each worker, that is, a metric that indicates how good is the worker at performing the labelling task. With this new information, the probability that a web page X is labelled by label L is estimated again. This cycle continues until convergence. See Figure 40.

– **Expectation maximisation** (EM): (Example lecture 9 slide 23-25). Iterates in two steps:

1. *E-Step*: estimate the labels ($P(x_j = l)$) from the answers of workers.

$$P(x_j = l) = \frac{1}{\sum_{i=1}^{N} w_i} \sum_{i=1}^{N} (w_i | a_i(x_i) = l)$$

2. *M-Step*: estimate the reliability of workers from the consistency of answers. $w_i$ = expertise of worker $i$ (initialized to $w = (1, 1, \ldots, 1)$), $M$=#web pages to label.

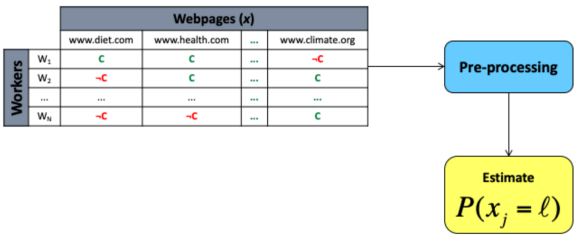


Figure 39: Non-iterative aggregate algorithm

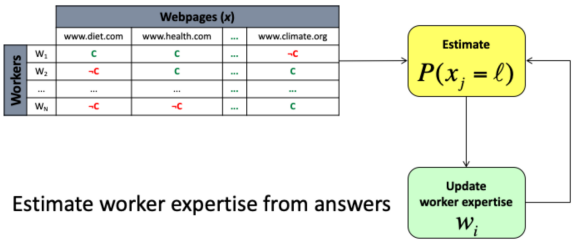


Figure 40: Iterative aggregate algorithm

We compute the percentage of correct answer (answer that the majority chose) for worker $i$ with this method.

$$w_i = \frac{1}{M}\sum_{j=1}^{M}(1|a_i(x_j) = \underset{l}{\operatorname{argmax}}\, P(x_j = l))$$

**Discretisation**: We have two main discretisation methods:

- **Unsupervised**: no class information → bins might confuse data from different classes.
  - Equal width: divide the range into predefined number of bins. In the case where the features values are not distributed uniformly, a large amount of important information can be lost after discretisation process (many very sparsely populated bins).
  - Equal frequency: divide the range into a predefined number of bins so that every interval contains the same number of values. Many occurrences of the same value could cause those occurrences to be assigned into different bins (non sense). This problem could be solved by merging neighbouring bins that contains duplicate values.
  - Clustering: Use any suitable clustering method for multi-dimensional data and assign one feature value per cluster. The advantage of that method is that it can be performed on all features at the same time, capturing in this way, possible interdependencies of features.
- **Supervised**: discretisation follows class boundaries. The idea is: if the class label does not depend on the choice among two (adjacent) intervals, the separation of the intervals does not provide useful information to the classifier. For that we can use the $\chi^2$ statistics (see Figure 41)

$$\chi^2 = \sum_{i=1,2}\sum_{j=1,2}\frac{(O_{ij} - E_{ij})^2}{E_{ij}}$$

  It is necessary to consult a $\chi^2$ distribution table with a degree of freedom = (#rows-1)*(#cols-1). If the value is greater than a threshold $p$ (typically 0.05 or 0.01), the two intervals are independent and can be merged. (See example lecture 9 slides 33-34 or check SciPy documentation with chisquare).

**Features selection**: Idea: discard irrelevant features. Reduce the number of N features from $N$ to $M$, $M < N$. There are $\binom{N}{M}$ possible subsets. Approaches:

- **Filtering**: Consider features as independent. Rank features according to their predictive power and select the best ones. Pros, cons:
  - \+ Independent of the classifier (performed only once)
  - \- Independent of the classifier (ignore interaction with the classifier)
  - \- Assumes features are independent (correlation ≠ causality) (e.g. see lecture 9 slide 42-43 for pitfall $\implies$ usage of wrapping)

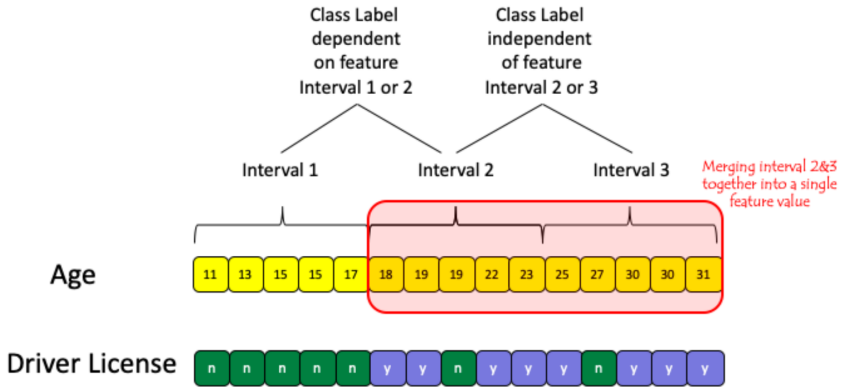


Figure 41: Example of supervised discretisation

| | Class 1 | Class 2 | *sum* |
|---|---|---|---|
| Interval 1 | $O_{11} = n_{11}$ | $O_{12} = n_{12}$ | R1 |
| | $E_{11} = \frac{R1 \cdot C1}{N}$ | $E12 = \frac{R1 \cdot C2}{N}$ | |
| Interval 2 | $O_{21} = n_{21}$ | $O_{22} = n_{22}$ | R2 |
| | $E_{21} = \frac{R2 \cdot C1}{N}$ | $E_{22} = \frac{R2 \cdot C2}{N}$ | |
| *sum* | C1 | C2 | N |

$O_{ij}$ observed frequency
$E_{ij}$ expected frequency

Table 7: Chi-square computation

Alternative approach: Information-theoretic (mutual information between feature $F$ and class label $C$) (See lecture 9 slide 40). *Note: The filtering approach tests whether a feature is dependent on the class label.*

- **Wrapper or Wrapping**: consider dependencies among features. Iteratively add features:
  - At each iteration create a classifier for each new feature and evaluate its performance
  - Add best feature or stop when no further improvement

  Pros, cons:

  \+ Interact with the classifier
  \+ No independence assumption
  \- Computationally intensive

  *Note: The wrapping eliminates strongly correlated features.*

**Feature normalisation**: Some classifiers do not manage well features with different scales (e.g. $10^7$ followers but 300 tweets). Features with large values dominate the others and the classifier tends to over-optimise them. Approaches:

- **Standardisation**: map to a normal distribution $N(0, 1)$ with

$$x'_i = \frac{x_i - \mu_i}{\sigma_i}$$

  where $\mu_i$ = mean value of feature $x_i$ and $\sigma_i$ = standard deviation. It makes the **assumption that the data has been generated by a Gaussian process (not necessarily true)!**

- **Scaling**: map to interval $[0, 1]$ with

$$x_i' = \frac{x_i - m_i}{M_i - m_i}$$

where $M_i$ and $m_i$ are the maximal and the minimal values of the feature $x_i$. If the data has outliers, they scale the "non-outliers" values to a very small interval.

### 2.4.3 Model training, selection and assessment

**Selecting performance metrics**: First part is model assessment: Having chosen a model, estimate the prediction error on new data (separate data into train vs. test). The simplest performance metric is the accuracy:

$$A = \frac{TruePositive + TrueNegative}{TP + TN + FP + FN} = \frac{TP + TN}{N}$$

It works when the classes are not skewed (e.g. lecture 9 slide 55). The next steps (precision, recall, F-score) has already been covered before in subsubsection 1.1.4.

**Model selection**: Estimating performances of different models to select the best one (classifier has some parameters to be tuned). We will have:

- Training data
- Validation data: used to select the best model (model selection)
- Test data: assess the performance of the finally resulting model (model assessment)

Model selection has multiples steps (search local minimum of loss function):

1. Split dataset into training and validation
2. Set classifier parameters
3. Train classifier with training set
4. Evaluate classifier with validation set
5. Performance acceptable ? take that model : goto 2

*Loss function*: $x_i$ indicates a feature and $y$ is the class label.

- categorical output (0-1 loss function) (= accuracy without normalisation) *please do not use that one*: $J = \sum_{i=1}^{n} \#(y \neq f(x_i))$
- real value output:
  - squared error: $J = \frac{1}{n} \sum_{i=1}^{n} (y_i \neq f(x_i))^2$
  - absolute error: $J = \frac{1}{n} \sum_{i=1}^{n} |y_i \neq f(x_i)|$

**Organizing training and validation set**: The simplest way to do model selection is constructing a training and validation set. Drawback: It does not use all the data to train, and it is possible that the validation set is not representative for the classification tasks (e.g. rare cases are missed (skewed distributions)). Fighting skew:

- **Stratification**: Select validation set as a random sample, but assure that each class is (approximately) proportionally represented
- **Over-sampling**: Including *over*-proportionally number from the *smaller* class

- **Under-sampling**: Including *under*-proportionally number from the *larger* class

**K-fold cross validation**: In order to address the drawback of holding out a simple validation set, a better approach is (see lecture 9 slide 70):

1. Split the training set into $k$ disjoint partitions, and train a model for each subset the dataset where one of the partitions has been removed as validation set ($\frac{k-1}{k}$ train, $\frac{1}{k}$ validation). It can also be used with k-fold cross-validation.
2. The model is evaluated against the validation set that has been held out.
3. Finally the performance as averaged over **all** $k$ **runs**.

This methods is a good compromise between having an unbiased estimate of the true accuracy and reasonable computation time. An extreme form of this algorithm is to set $k = N$ where $N$ is the number of item in the dataset.

**How good is a model?**: Model is a function $f_D$ that estimates a function

$$f : X^d \to Y \text{ with } y = f(X)$$

The error is evaluated by a squared error measure:

$$Err(f_D, T) = \frac{1}{|T|} \sum_{X \in T} (f_D(X) - y)^2$$

where $D$ = training set and $T$ = validation set (on which error is evaluated i.e. test set). We can compute:

- train error: $Err_{train} = Err(f_D, D)$
- test error: $Err_{test} = Err(f_D, T)$

Repeatedly evaluate error for different models generated from different training sets $D \in \mathcal{D}$ and corresponding test set $T(D)$. We obtain:

- expected train error: $EErr_{train} = E_{\mathcal{D}}(Err(f_D, D)) = \frac{1}{|\mathcal{D}|} \sum_{D \in \mathcal{D}} Err(f_D, D)$
- expected test error: $EErr_{test} = E_{\mathcal{D},T}(Err(f_D, T(D))) = \frac{1}{|\mathcal{D}|} \sum_{D \in \mathcal{D}} Err(f_D, T(D))$

The error can be rewritten as $EErr_{test} = Bias^2 + Variance$ where

- $Bias = E_{\mathcal{D},T}[f_D(X) - y]$
- $Variance = E_{\mathcal{D},T}[(f_D(X) - \bar{f}(X))^2]$ with $\bar{f}(X) = E_{\mathcal{D}}[f_D(X)]$

There is usually a bias-variance tradeoff caused by model complexity (e.g. lecture 9 slide 81). See also Figure 42.

- **Complex models** (many parameters) usually have lower bias, but higher variance $\implies$ **over-fitting**
- **Simple models** (few parameters) have higher bias, but lower variance $\implies$ **under-fitting**

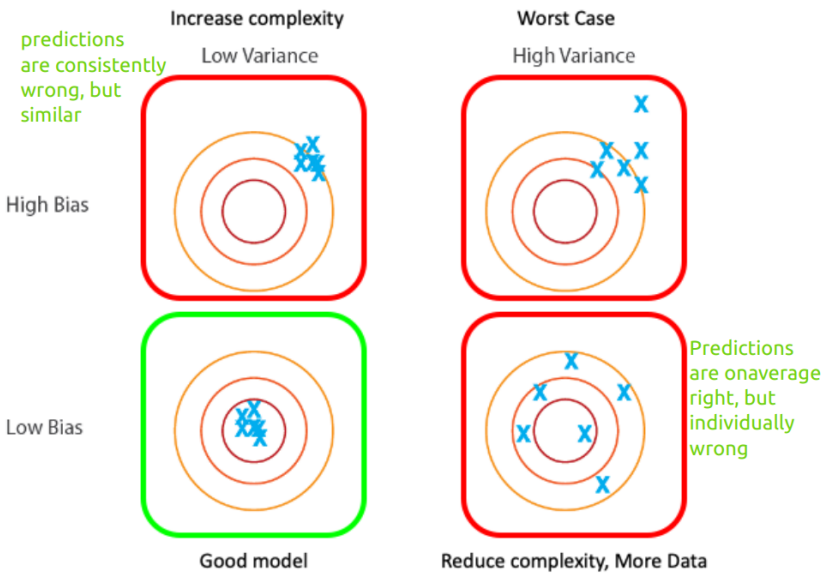


Figure 42: Bias and variance (lect 9 slide 84)

## 2.5 LEC10 : Applied Classification

### 2.5.1 Document classification

Task: Given a training set of labelled documents, construct a classifier that decide for an unlabelled document which label it has. A document classifier has many features:

- Words of the documents
  - bag of words
  - document vector
- More detailed information on words
  - phrases (2+ adjacent words)
  - word fragments (e.g. "more than" → ["mor", "ore", "tha", "han"])
  - grammatical features (classify each words into [adjective, adverb, noun, number, verb, ...])
- Any metadata know about the document and its author (e.g. username).

Document classification can be used for spam filtering, sentiment analysis, document search, ...

**Simple method: k-nearest-neighbors (kNN)**: Use vector space model. To classify a document $D$:

1. retrieve the $k$ nearest neighbors in the training set according to the vector space model.
2. Choose the majority class label as the class of the document.

In the context of kNN a model is complex when k is small, since for every document very different sets of neighbors are chosen. Correspondingly it holds that for small k (complex model) we have low bias, but high variance and for large k we have high bias and low variance. For kNN the training cost is low but the testing is expensive.

**Naïve Bayes classifier**: Apply Bayes law.

- Feature: Bag of words model: all words and their counts in the document. By Bayes law the probability that a document $D$ has the class $C$ becomes:

$$P(C|D) \propto P(C) \prod_{w \in D} P(w|C)$$

For this method to work, words not occurring in the training set, but in the test set, need to have non-zero probabilities.

- Training: How characteristic is a word $w$ for class $C$? ("+1" are called Laplacian smoothing (non-zero prob.))

$$P(w|C) = \frac{|w \in D, D \in C| + 1}{\sum_{w'} |w' \in D, D \in C| + 1}$$

How frequent is class $C$?

$$P(C) = \frac{|D \in C|}{|\mathcal{D}|}$$

- Classification: Thus the most probable class is the one with largest probability.

$$C_{NB} = \underset{C}{\text{argmax}}(\log P(C) + \sum_{w \in D} \log P(w|C))$$

**Classification using word embeddings**(e.g. Fasttext): Here the neighbors of a word in a sentence will also represent the word (e.g. see lecture 10 slide 9). Probability that word $w$ occurs with context word $c$:

$$P(D = 1|w, c; \theta) = \frac{1}{1 + e^{-v_c \cdot v_w}}$$

Now consider (class, paragraphs) pairs instead of (word, context) pairs. Word $w \rightarrow$ class label $C$ ; context words $c \rightarrow$ paragraph $p$.

$$\text{Learn } P(C|p) = \frac{e^{v_p \cdot v_C}}{\sum_{C'} e^{v_p \cdot v'_C}} \text{ (SGD)}$$

Paragraph feature: $v_p = \sum_{w \in p} v_w$. Then predict label from paragraph features.

### 2.5.2 Recommender systems (RS)

Given a *user* model and a set of *items*, a recommender system is a function that helps to match users with items by *ranking* the items in order of decreased *relevance*. There exists different classes of recommender systems.

- **Collaborative RS**: "Tell me what *other people* like". See Figure 43.
  - Based on **collaborative filtering**. A widely used approach to generate recommendations. Approach (*Wisdom of the crowd*): Users give rating to items. Users with similar tastes in the past will have similar tastes in the future.
  - Basic technique of **user-based collaborative filtering**: Given a user $x$ and an item $i$ not rated by $x$, estimate the rating $r_x(i)$ by
    1. Finding a set of users $N_U(x)$ who rated the same items as $x$ (= neighbors of $x$) in the past **and** have rate $i$
    2. Aggregate the ratings of $i$ provided by $N_U(x)$

    Compute the ratings for all the items not rated by $x$ and recommend the best-rated ones. See Figure 44. Now we need to define:
    * A metric to compute similarity between users

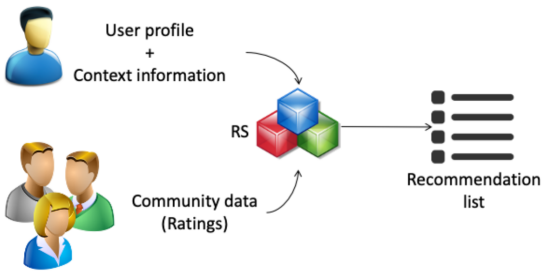


Figure 43: Collaborative recommender system

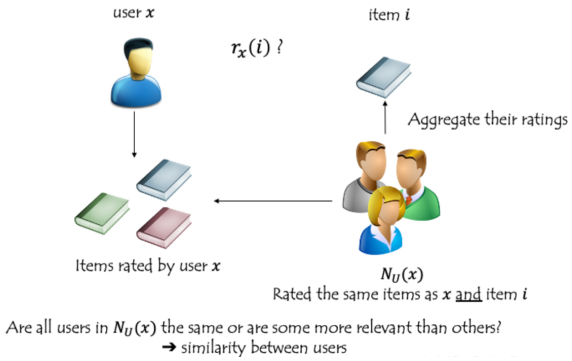


Figure 44: User based collaborative filtering

  - The number of neighbours considered for $N_U(x)$
  - A way to aggregate the ratings of I provided by $N_U(x)$
- **Similarity between users**: (example and many details in lecture 10 slide 23 in text)
  - *Pearson correlation coefficient*: Widely used. It returns a value between $-1$ and 1.

$$sim(x,y) = \frac{\sum_{i=1}^{N}(r_x(i)-\bar{r_x})(r_y(i)-\bar{r_y})}{\sqrt{\sum_{i=1}^{N}(r_x(i)-\bar{r_x})^2}\sqrt{\sum_{i=1}^{N}(r_y(i)-\bar{r_y})^2}}$$

  - *Cosine similarity*: It returns a value between 0 and 1. It follows exactly the same principles as for document similarity in vector space retrieval.

$$sim(x,y) = \frac{\sum_{i=1}^{N} r_x(i)\cdot r_y(i)}{\sqrt{\sum_{i=1}^{N} r_x(i)^2}\sqrt{\sum_{i=1}^{N} r_y(i)^2}}$$

    where
    - $x$, $y$: users
    - $N$: number of items $i$ rated by both $x$ and $y$
    - $r_x(i)$: rating of user $x$ of item $i$
    - $\bar{r_x}$: average ratings of user $x$
- **Aggregation function** (user-based):

$$r_x(a) = \bar{r_x} + \frac{\sum_{y\in N_U(x)}}{\sum_{y\in N_U(x)} sim(x,y)(r_y(a)-\bar{r_y})|sim(x,y)|}$$

  where $N_U(x)$ = neighbours of user $x$ and $a$ = item not rated by $x$
- Problems of user-based collaborative filtering:
  - cold start: users or items without ratings
  - scalability: large numbers of users
  - data dispersion: highly variable ratings, difficult to find similar users.

  Possible solution: Item-based collaborative filtering
- **Item-based collaborative filtering**: Item-based collaborative filtering does not solve the cold-start problem but has better scalability. (See lecture 10 slide 34 for a class question on cold start). See Figure 45 and Figure 46.
- Basic technique of **item-based collaborative filtering**: Use the similarity between items (and not users) to make predictions. Given a user $x$ and an item $i$ not rated by $x$, estimate the rating $r_x(i)$ by:

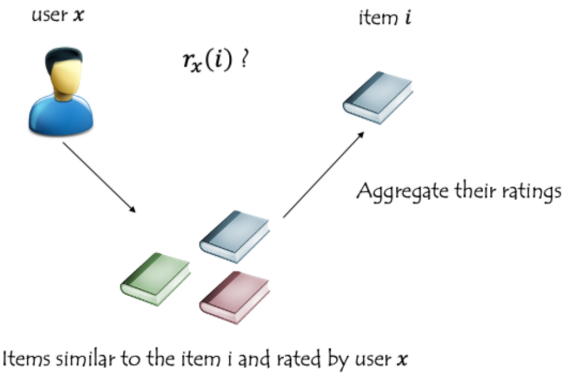


Figure 45: Item based collaborative filtering

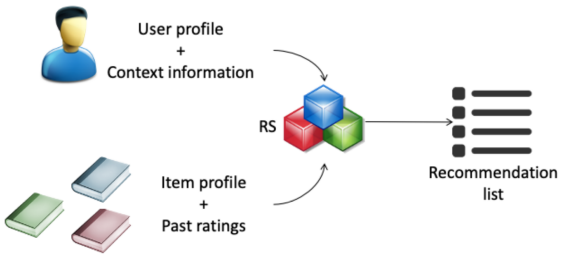


Figure 46: Content-based recommender system

1. Find a set of items $N_I(i)$ which are similar to $i$ and that have been rated by $x$
2. Aggregate the ratings of the items in $N_I(i)$ and use the aggregation as prediction of $r_x(i)$

Now we need to define:

* A metric to compute similarity between items
* The number of neighbours considered in $N_I$
* A way to aggregate the ratings of the items in $N_I$

– **Aggregation function** (item-based):

$$r_x(a) = \frac{\sum_{b \in N_I(a)} sim(a,b) r_x(b)}{\sum_{b \in N_I(a)} |sim(a,b)|}$$

where $N_I(a)$ = neighbours of item $a$ and $b$ = item rated by $x$

– Item-based collaborative filtering does not solve the cold-start problem but has better scalability. (See lecture 10 slide 34 for a class question on cold start).

1. Calculate all the pair-wise item similarities in advance
2. Items similarities are more stable
3. The neighbourhood $N_I(a)$ (composed of the other items rated by user x) to be considered at runtime is small.

- **Content-based RS**: "Show me more of what *I* liked". The collaborative filtering methods (seen before) use only ratings but do not exploit any other information about the items. However the *content* of the item might provide some useful information! Then we don't need a community of users any more!

  – Basic technique of **content-based RS**: Given the items that have been rated by user $x$ in the past:

  1. Find the items that are similar to the past items
  2. Aggregate the ratings of the most similar items

  We need to define:

  1. A way to formalize the item content
  2. A similarity measure between items
  3. An aggregation function for the ratings

- Most methods originate from information retrieval to extract the content of an item could be used here. For example, given an item and a textual description (e.g. movie + synopsis, book review, ...), compute the TF-IDF weights.

$$w(t,a) = tf(t,a) \cdot idf(t) = \frac{freq(t,a)}{\max_{s \in T} freq(s,a)} \cdot \log(\frac{N}{n(t)})$$

  where $N$ = number of items to recommend and $n(t)$ = number of items where term $t$ appears.
- The cosine similarity in content-based recommendation considers the items $a$ and $b$ as two $T$-dimensional vectors, where each dimension is the TF-IDF measures of a specific term $t$.
- After TF-IDF + cosine similarity, the simplest approach to estimate the rating of item $a$ still not rated by user $x$ is finding $k$ nearest neighbours of item $a$ that have already been rated by user $x$ and aggregate their ratings. **Approach**: Given a set of items $D$ that have been rated by the user, find the nearest neighbours $N_I(a)$ in $D$ of a new item $a$. Use the ratings of the nearest neighbours to predict the rating of $a$ with the "item-based aggregation function" seen before.
- Pros, cons:
  + Scalable: tf-idf of items can be computed offline
  - Cold start problem for users with no ratings
  - Content can be limited or impossible to extract (e.g. multimedia)
  - Tends to recommend "more of the same"

- We always (for user-, item-based collaborative or content-based RS) have the same problem. For a user that has not done any ratings (e.g. new user), we cannot make a prediction. However for an item that has not received any ratings, only content-based RS is able to make a prediction.

- **Matrix factorization**: There exists some advanced techniques such as matrix factorization (see lecture 10 slide 43) that are a combination of different techniques. For example, matrix factorization transforms the user-item rating matrix into a latent factor model, where each user and item is described as a vector in a k-dimensional space. It can be understood as a combination of user-based and item-based collaborative filtering. This approach is analogous to singular value decomposition for document retrieval. However, a direct application of SVD is not possible because the user-item matrix is not complete (missing ratings).
  - **Factorization problem**: Rating matrix $R$ with dimension $|U| \times |D|$ (users $U$ and items $D$). The goal is to decompose $R$ (approximately):

$$R \approx P \times Q^T = \hat{R}$$

    $K$ latent factors
    * $P$ is a $|U| \times K$ matrix: user features (it's a mapping Users to latent space)
    * $Q$ is a $|D| \times K$ matrix: item features

    Problem: $R$ has many undefined values (missing ratings)!
  - **Approximation error**: For a given rating $r_{ij}$:

$$e_{ij}^2 = (r_{ij} - \hat{r}_{ij})^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2$$

Minimizing error: compute gradient (for SGD)

$$\frac{\partial}{\partial p_{ik}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij}) q_{kj} = -2 e_{ij} q_{kj}$$

$$\frac{\partial}{\partial q_{kj}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij}) p_{ik} = -2 e_{ij} p_{ik}$$

- **Stochastic gradient descent (SGD)**: Update rule: for some random pair $i, j$:

$$p_{ik} := p_{ik} - \alpha \frac{\partial}{\partial p_{ik}} e_{ij}^2 = p_{ik} + 2\alpha e_{ij} q_{kj}$$

$$q_{kj} := q_{kj} - \alpha \frac{\partial}{\partial q_{kj}} e_{ij}^2 = q_{kj} + 2\alpha e_{ij} p_{ik}$$

  Repeat until error is small. Needs only to be **computed for ratings $r_{ij}$ that exist**!
- **Regularization**: In order to **avoir overfitting**, usually a regularization term is added and the update rule is modified correspondingly.

$$e_{ij}^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2 + \lambda(\|P\|^2 + \|Q\|^2)$$

  and then

$$p_{ik} := p_{ik} - \alpha \frac{\partial}{\partial p_{ik}} e_{ij}^2 = p_{ik} + 2\alpha(e_{ij} q_{kj} - \lambda p_{ik})$$

$$q_{kj} := q_{kj} - \alpha \frac{\partial}{\partial q_{kj}} e_{ij}^2 = q_{kj} + 2\alpha(e_{ij} p_{ik} - \lambda q_{kj})$$

### 2.5.3 Mining social graphs

Up to now we considered objects described by attributes and the relationships among objects was inferred from distance measure on attributes. In many cases explicit relationships are given. Explicit relationships = graphs (e.g. social network graphs). Graphs structure can also be inferred from distance measure (e.g. for documents) (e.g. There the relationships among users are given by different friend relationships or interactions such as likes and retweets). We will consider the case of unweighted graphs. Graphs often contain structure: clusters (also called communities, modules).

**Social network analysis**: Clusters in social networks related to interests or level of trust. The community structures might be useful in different domains (e.g. job recommendations, detection of media influencers, ...).

The task is to find densely linked clusters. The goal of a community detection algorithm is to identify communities that are heavily intra-linked (high intra-cluster similarity) and scarcely inter-linked (low inter-cluster similarity).

*Note: Cliques is an algorithm that could detect some communities but not all.*

There exists two types of community detection algorithms. We will present one algorithms of each category:

- **Agglomerative algorithms** *merge* nodes and communities with high similarity. (**Louvain algorithm**)
- **Division algorithms** *split* communities by removing links that connect nodes with low similarity. (**Girvan-Newman algorithm**)

Let's begin with **Louvain modularity algorithm**.

- This is an agglomerative community detection:
  - Based on a measure for community quality (**modularity**)
  - Greedy optimisation of modularity
- The Louvain method is widely used in social network analysis and beyond. It extract communities from very large networks very fast
- **Overall algorithm**:
  1. Initially every node is considered as a community.
  2. The communities are traversed, and for each community it is tested whether by joining it to a neighboring community, the modularity of the clustering can be improved.
  3. This process is repeated till no new communities form anymore.
  4. Complexity: $O(n \log n)$. (traverse everything $\implies O(n)$, build the tree and use that tree $\implies O(\log n)$)

  - **Measuring community quality (modularity)**: Communities are sets of nodes with many mutual connections, and much fewer connections to the outside. **Modularity measures this quality: The higher the better**. Thus, we want to optimize it.
    $$\sum_{C \in \text{communities}} (\#\text{edges within } C - \text{expected } \#\text{edges within } C)$$
  - Expected number of edges: Graph with unweighted edges:
    * $m$ = total number of edges $\implies \exists\ 2m$ "edge ends"
    * $k_i$ = number of outgoing edges of node $i$ (degree)
    * A node $i$ distributes $k_i$ edges ends uniformly to other nodes. Will node $j$ receive the edge end from $i$? Node $j$ receives fraction of $\frac{k_j}{2m}$ edge ends from each other nodes. It implies that the number of expected edges connecting nodes $i$ and $j$ would be $\frac{k_i k_j}{2m}$
  - **Modularity** measure $Q$:
    * $A_{ij}$ = effective number of edges between nodes $i$ and $j$
    * $C_i, C_j$ = communities of nodes $i$ and $j$

    $$Q = \frac{1}{2m} \sum_{i,j} (A_{ij} - \frac{k_i k_j}{2m}) \delta(C_i, C_j)$$

    Properties:
    · The $\delta$ function assures that only nodes belonging to the same community are considered, since it returns 1 when $C_i = C_j$
    · $Q \in [-1, 1]$
    · $\alpha < Q$ (where $\alpha \in [0.3, 0.7]$ is a threshold) means significant community structure
  - *Note: Modularity can also be used to evaluate the best level to cutoff a hierarchical clustering.*
  - *Note: Modularity clustering will end up with a single community at top level only for connected graphs.*
  - *Note: Modularity will end up with the same community structure if we use the same ordering between executions.*

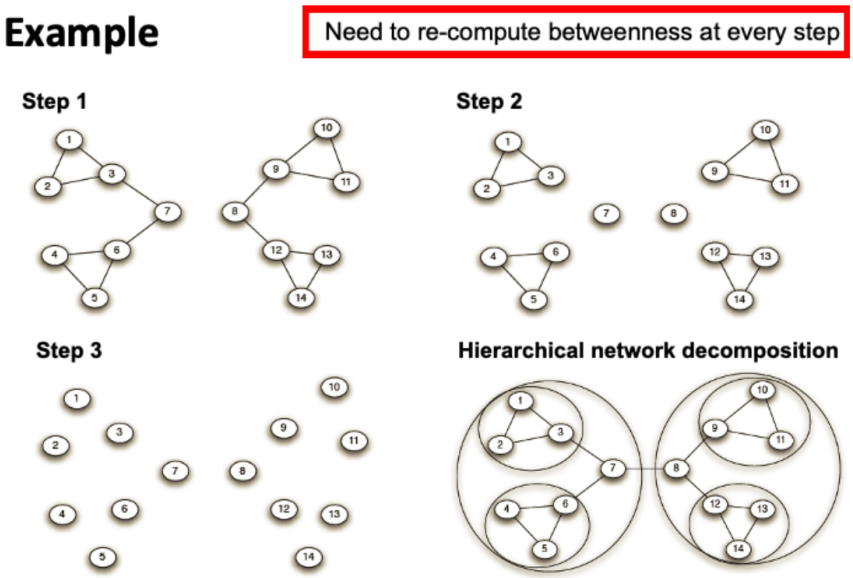


Figure 47: Example of Girvan-Newman algorithm

- Locally optimize communities: What is the modularity gain by moving node $i$ to the communities of any of its neighbors? You have to test all possibilities and choose the best (larger modularity gain). (See example in lecture 10 slide 62-65).
- Now that all nodes have been processed, we merge nodes of the same communities in a single new nodes and restart processing. (See example lecture 10 slide 66).

Now take a look into the **Girvan-Newman algorithm**.

- This is a divisive community detection:
  - Based on **betweenness measure** for edges. It gives an indication which edges are likely to connect different communities, and thus are good splitting points, to partition larger parts of the network in to communities.
  - Decomposition of network by splitting along edges with highest separation capacity.
- **Overall algorithm**: See example in Figure 47
  1. Repeat until no edges are left:
     (a) Calculate betweenness of edges
     (b) Remove edges with highest betweenness
     (c) Resulting connected components are communities.
  2. Results in hierarchical decomposition of network.
- **Edge betweenness**: fraction of number of shortest paths passing over the edge. (See example lecture 10 slide 73).

$$betweenness(v) = \sum_{x,y} \frac{\sigma_{xy}(v)}{\sigma_{xy}}$$

where

  - $v$ is a node.
  - $\sigma_{xy}$ = number of shortest paths from $x$ to $y$

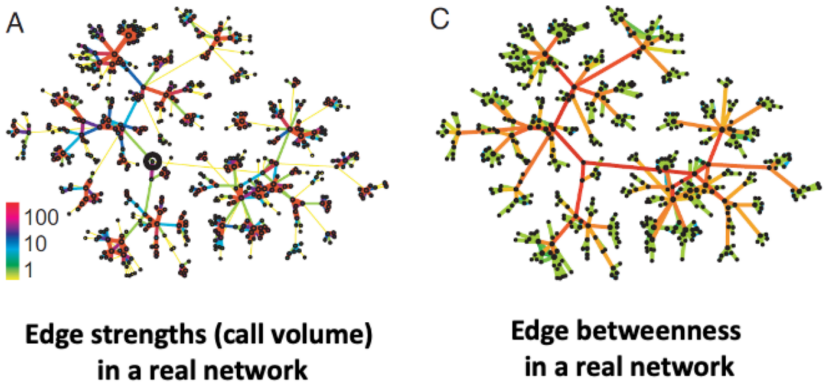


Figure 48: Intuitive example of edge betweenness

- $\sigma_{xy}(v)$ = number of shortest paths from $x$ to $y$ passing through $v$.

Alternatively, there exist also the concept of *random-walk betweenness*. A pair of nodes $m$ and $n$ are chosen at random. A walker starts at $m$, following each adjacent link with equal probability until it reaches $n$. Random walk betweenness $x_{ij}$ is the probability that the link $i \to j$ was crossed by the walker after averaging over all possible choices for the starting nodes $m$ and $n$.

To see an intuitive example about edge betweenness, see Figure 48.

- **Computing betweenness**: To compute betweenness of paths starting at node $A$. (See schemes lecture 10 slides 78-81 or simply exercise session 09.Social_Network_Analysis_sol).

  Scheme or ref. to lect?

  1. *BFS*: First step is to perform BFS starting from $A$.
  2. *Path counting*: Count the number of shortest paths from $A$ to all other nodes of the network.

$$\#\text{shortest\_paths}(A \leftrightarrow X) = \sum_{P \text{ parent of } X} \#\text{shortest\_paths}(A \leftrightarrow P)$$

  3. *Edge flow*: **See lecture 10 slide 81**
     - 1 unit of flow from $A$ to each node
     - Flow to be distributes evenly over all paths

$$\text{weight\_to\_distribute}(X) = 1 + \sum_{C \text{ child of } X} \text{edge\_weight}(X \leftrightarrow C)$$

     where

$$\text{edge\_weight}(X \leftrightarrow C) \text{ depends on ratio of } \#\text{shortest\_paths}(A \leftrightarrow X)$$

     - Sum of the flows from all nodes equals the betweenness value
  4.

**Algorithm for computing betweenness**:

1. Build one BFS structure for each node
2. Determine edge flow values for each edge using the previous procedure
3. Sum up the flow values of each edge in all BFS structures to obtain betweenness. Flows are computed between each pairs of nodes $\implies$ final values divided by 2.

- **Discussion**:
  - Classical method works for smaller networks (Grivan-Newman algorithm is not so scalable).
  - Complexity:
    * Computation of betweenness for one link: $O(n^2)$
    * Computation of betweenness for all link: $O(Ln^2)$
    * Sparse matrix: $O(n^3)$ (If we assume sparse networks, where the number of links is of the same order as the number of nodes, the total cost is cubic.)
  - Its complexity motivates the creation of modularity seen before.

## 2.6 LEC11 : Semantic Web

**Implicit vs. explicit knowledge**: Information retrieval and data mining aim at extracting and using knowledge that is implicitly represented in documents.

### 2.6.1 Semi-structured data

**Schemes** define data structures for databases (relational database schemes, XML). It is an agreement on data structures (keep data consistent).

- HTML has never been conceived to specify semantics of data (then it is not a good way to structure data). Limitations of HTML:
  - Structure of data expressed as layout (e.g. <tr><td> leech </td><tr>)
  - Semantics of data hard to analyse and difficult to share
  - No schemes, no constraints

  Making the meaning of data explicit (embedding of schema information into the data):
  - Website A: <Species> leech </Species>
  - Website B: <organism> leech </organism>

  In the Web this approach has found wide-spread adoption through the XML format. XML generalizes the markup approach from HTML to allow applications to specify their application-specific tags and to embed them into documents.

**Semi-structured data**: Data that embeds schema information (through tags, markup, etc.) to explain the meaning of data values and to relate different data values (e.g. hierarchically). No predefined data structure, no schema required! (example of semi-structured data: email, JSON, XML, ...). Semi-structured data always embeds schema information into the data. *Note: The duality of XML, being a document model (It has a serialized representation (a string representation that describes a structured data)) and a data model at the same time makes it particularly suitable for applications that have to exchange data over communication networks.*

**Schema-less data**: pros and cons:

+ Increased flexibility (e.g. dynamically adding or dropping structural elements such as attributes).

+ Self-contained data (e.g. context (schema information) directly encoded into data (markup))

- Loss of consistency

- Certain optimisations not feasible

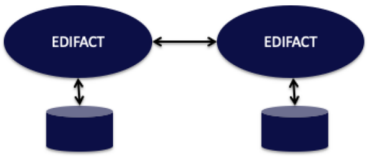


Figure 49: Standardization

Figure 50: Mapping

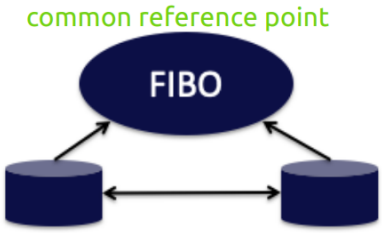


Figure 51: Ontologies

**Serialized vs. semi-structured data**: Many of the popular semi-structured data models have a serialized format (XML) or have been conceived to serialize data (JSON). However, there exist semi-structured data models that are not based on a serialized representation → knowledge graphs.

### 2.6.2 Semantic web

User-defined markup (schemes): provides possibility to share interpretation of data across various applications. The problem is that semantic heterogeneity (The same concepts can be represented in different application contexts differently (e.g. using different terms to denote the same meaning)) implies semantic web.

Three ways to overcome semantic heterogeneity:

- **Standardization** : agree on common user-defined markup (schemes). This approach is used when there exists already (historically) a wide agreement on the structure of relevant data and their interpretation.
  - Great if no pre-existing applications
  - Great if power player enforces it
- **Translation**: create mappings among different schemes and databases. This is the approach that has been extensively studied for data integration problems in relatively small and controlled domains, such as inside businesses and organizations.
  - requires human interpretation and reasoning
  - mappings can be difficult, expensive to establish
- **Annotation**: create relationships to agreed upon conceptualizations. This one is slightly different from the second: instead of engineering mappings between heterogeneous schemes for each integration problem, one first agrees on a common conceptualization of the domain, covering relevant aspects for a large class of applications.
  - requires human interpretation and reasoning
  - annotation can be difficult, expensive to establish
  - reasoning over the conceptualization can provide added value

Conceptually there exists three main possibilities of how to address semantic heterogeneity.

1. Standardization: mapping through standards. See Figure 49
2. Mapping: direct mapping. See Figure 50.

   The problem of creating mappings among information systems is known as the *schema mapping* problem. Assume all data represented in canonical data model (e.g. relational)
   - detect correspondences (schema matching)

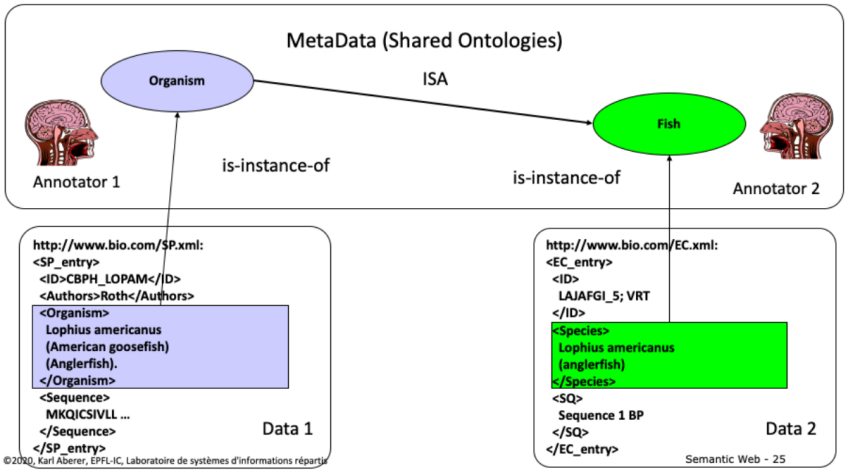


Figure 52: Example of mediated mappings (ISA = "... is a ...")

- resolve conflicts!
- integrate schemes (schema mapping)

Mapping are frequently expressed as queries (e.g. SQL queries). Tools for supporting these steps are called schema mapping tools.

3. Ontologies: mediated mapping. Ontologies are an explicit specification of conceptualization of the real world. See Figure 51 and Figure 52.

   - Ideally:
     - different information systems agree on the same ontology (See Figure 52. If everyone use same ontology, then everyone can say that a fish IS An organism).
     - relate their model/schema/data elements to the ontology
     - mapping can be constructed via the ontology

     It requires agreement on the conceptualization of the real world!
   - Creating ontologies: Different approaches to create ontologies:
     (a) Ontology engineering
       - Manual effort
       - Tools for editing and checking consistency

     (b) Automatic generation of ontologies: From large document collections or existing structured data sources.
   - Modeling and encoding of ontologies issues:
     - Modeling primitives and their semantics
       * What does an arrow mean?
       * What does "instance-of" mean?
       * What does ISA mean?
     - Standardized encodings of the model. The encoding is equally important, as it should be done in a standardized form. If this were not the case we would immediately loose the advantage of having a common conceptualization of the world at the abstract level, since we were not able to exchange it properly with others.
       * Into document language (e.g. XML, HTML)
       * Enriching document content with semantic markup

| | HTML | XML | RDF | OWL |
|---|---|---|---|---|
| Simplicity | + | + | + | + |
| Exchangeability | + | + | + | + |
| Non-intrusive annotation | + | - | + | + |
| Domain-specific vocabularies | - | + | + | + |
| Modeling primitives | - | - | + | + |
| Reasoning capabilities | - | - | - | + |
| | Separate structure from presentation | | | |
| | | Separate interpretation from structure | | |

Table 8: Model requirements for ontologies (lect 11 slide 32 for details)

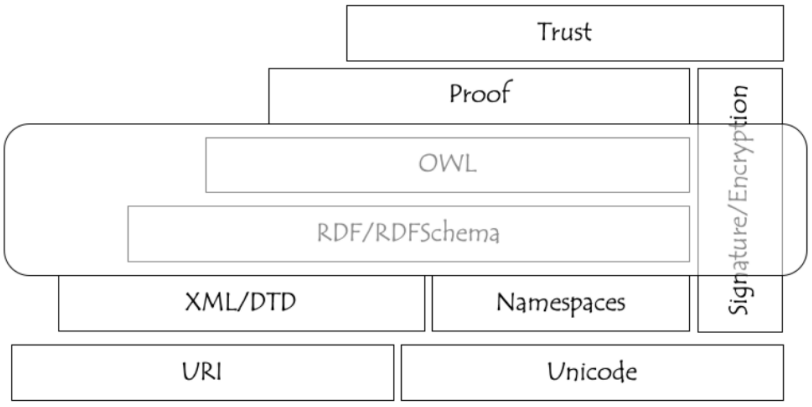


Figure 53: W3C web architecture

- With respect to modeling and encoding of ontologies for the Semantic Web, there exist a number of requirements. See Table 8.
- The Semantic Web standards RDF and OWL are positioned in the Semantic Web architecture in top of the syntactic layer. See Figure 53.

### 2.6.3 Resource description framework (RDF)

The relation RDF vs RDFS is comparable to well-formated vs valid XML.

- **RDF (instances)**
  - Statements about resources (addressable by an URI) and literals (XML data)
  - Statements are of the form: subject property object
  - Like simple natural language sentences
  - RDF statements are themselves resources (reasoning about RDF)
  - Properties define relationships to other resources or atomic values
- **RDF-schema (RDFS)**. See Figure 54
  - Data model to specify schemes for RDF instances
  - Which properties are applicable for which objects with which subjects
  - Defines "grammar" and "vocabulary" for semantic domains (ontology language)

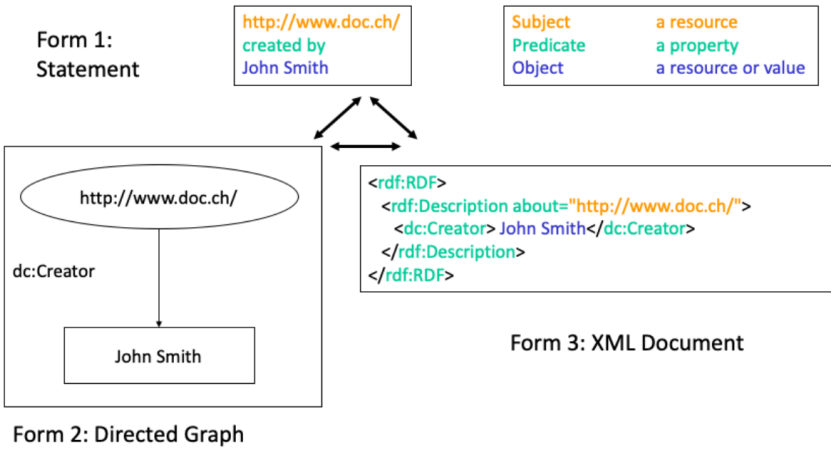


Figure 54: RDFS example

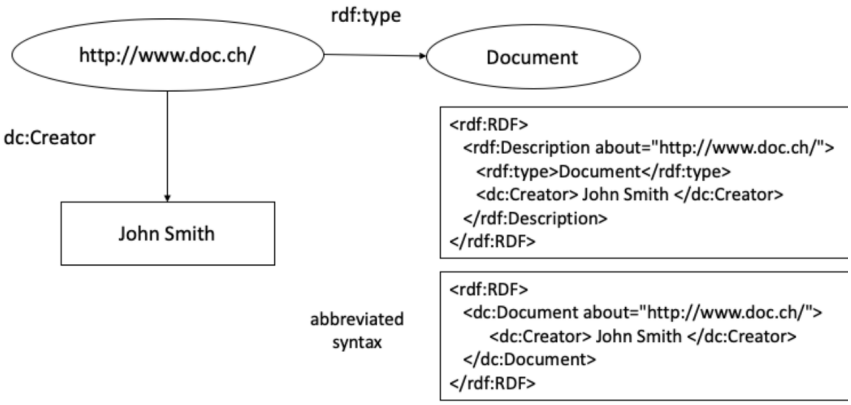


Figure 55: Typing resources with *rdf:type*

- **Typing resources**: Resources can be associated with a type by using *rdf:type* (a special property). See Figure 55
- **Complex values**: Use an intermediate resource. See lecture 11 slide 40 for an example. Instead of using:

```
<rdf:Description  about="http://www.doc.ch">
    <dc:Creator> Jane Doe </dc:Creator>
</rdf:Description>
```

  We could use:

```
<rdf:Description  about="http://www.doc.ch">
    <dc:Creator>
        <rdf:Description about="Person://1234/1">
            <name> Jane Doe </name>
            <email> janedoe@epfl.ch </email>
        </rdf:Description>
    </dc:Creator>
</rdf:Description>
```

- **RDF containers**:
  - Containers

        * Bag (unordered)
        * Seq (ordered)
        * Alt (alternatives)
    - Quantifiers
        * about: John is author of the talk (consisting of many slides)
        * aboutEach: John is author of each slide of the talk

    Example about container Seq:

```
<rdf:RDF>
    <rdf:Description about="/talks/7">
        <s:slides>
            <rdf:Seq>
                <rdf:_1 resources="/slides/42"/>
                <rdf:_2 resources="/slides/73"/>
            </rdf:Seq>
        </s:slides>
    </rdf:Description>
</rdf:RDF>
```

    If the order was not important, we might use *rdf:li* instead of *rdf:_1* and *rdf:_2*.

- **Creating new resources**: New RDF resources are created by using *rdf:ID* (a special property). This attribute replaces *rdf:about* attribute when the statement is about a new resource instead about an existing resource. Using the identifier, this new resource can then be referred to in other RDF statements. A resource is in general anything that can be identified on the Web. Let see an example:

```
# Alternative syntax
<rdf:RDF>
    <rdf:Description about="#12345">
        <rdf:type> Document </rdf:type>
        <dc:Title> Harry Potter </dc:Title>
    </rdf:Description>
</rdf:RDF>

# Abbreviated syntax
<rdf:RDF>
    <rdf:Description rdf:ID="12345">
        <rdf:type> Document </rdf:type>
        <dc:Title> Harry Potter </dc:Title>
    </rdf:Description>
</rdf:RDF>

# Abbreviated syntax version 2
<rdf:RDF>
    <dc:Document rdf:ID="12345">
        <dc:Title> Harry Potter </dc:Title>
    </dc:Document>
</rdf:RDF>
```

- *Note: A basic statement in RDF would be expressed in the relational data model by a table with two attributes.*

- *Note: The type statement in RDF would be expressed in the relational data model by a table with one attribute.*

- **RDF reification**: In RDF everything is a resource. In particular, RDF statements are considered as resources and therefore it should be possible to make statements about them. When considering the graph representation of RDF, it is not immediately clear of how to treat a statement as a resource, since a statement consists of three structural elements, the subject, the object, and the predicate. But we can apply the same "trick" as we did already for complex objects and collections. We introduce a new resource which serves as representative for the statement. This resource obtains as properties all the constituents that make up the statement it represents. This process is called reification.

  Syntax:

```
<rdf:RDF>
    <rdf:Description about="#triple123">
        <rdf:subject resource="http://www.doc.ch"/>
        <rdf:predicate resource="http://description.org/schema#Creator"/>
        <rdf:object> John Smith </rdf:object>
        <rdf:type resource="http://www.google.ch"/>
        <dc:Creator> Jane Doe </dc:Creator>
    </rdf:Description>
</rdf:RDF>
```

- *Note: Reified statements always make a statement about another statement.*

We will now talk about RDFS.

- **RDF Schema (RDFS) - Classification**: We now introduce RDF Schema. RDF Schema provides two basic mechanisms. See Figure 56 and lecture 11 slide 48 for more details.

  1. Categorization RDF resources, into *classes*. Classes allow to associate a type with an RDF instance. Different classes can be in a subClass relationship (be included in each other). Classes are themselves RDF instances.
  2. Constraints on the possible use of properties, in the form of constraints expressing which classes can participate as subjects and object in statements using a specific property.

- **RDFS - Properties**: Connect resources. See Figure 57 and lecture 11 slide 50 for the RDFS syntax.

  – The RDF instance must have the properties that are declared for the class
  – *rdfs:domain*: classes of which the instances may have a property
  – *rdfs:range*: classes of which the instances may be the value of a property

### 2.6.4 Semantic web resources

Examples of popular ontologies and knowledge bases:

- WordNet: English dictionary with semantic relationships

  – Synonymy: Words that have similar meaning (e.g. happy, glad)
  – Antonymy: The opposite of synonymy (e.g. happy, sad)
  – Nouns only:

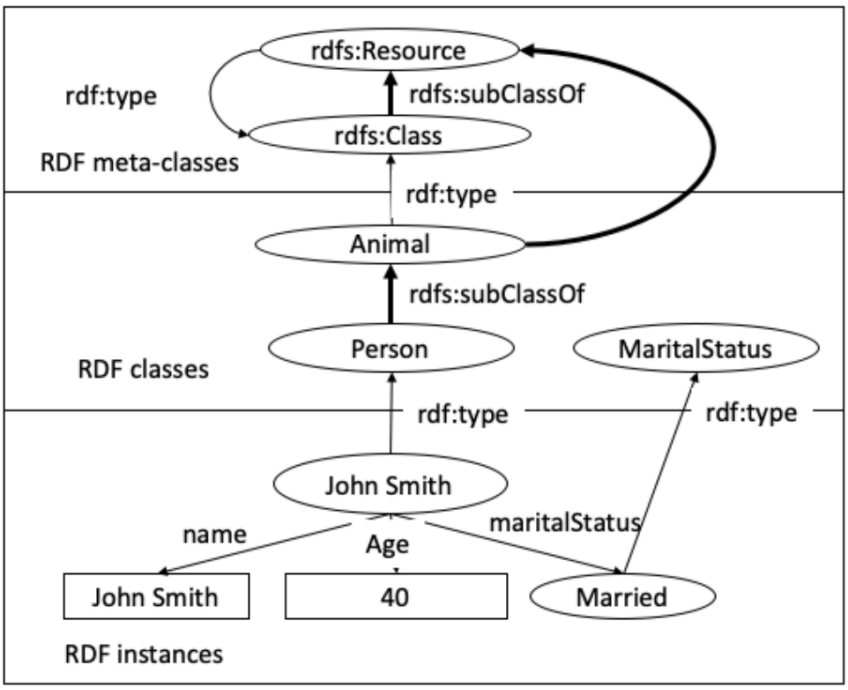


Figure 56: RDFS classification

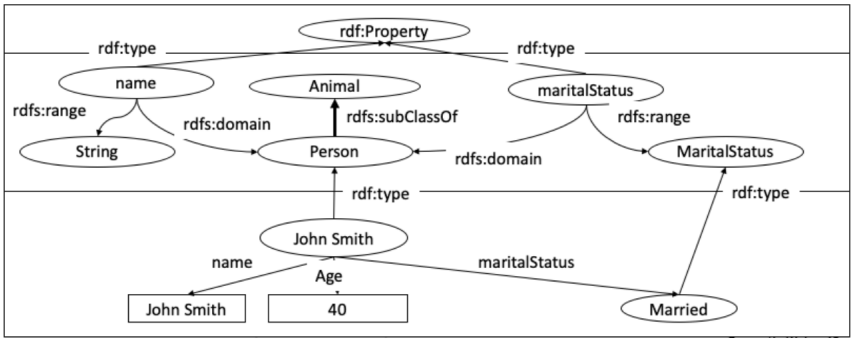


Figure 57: RDFS properties, check lecture 11 slide 50 for syntax

    - Hypernymy: hierarchical relationship between words (e.g. furniture is a hypernymy of chair since chair is a piece of furniture)
    - Hyponymy: Opposite of hypernymy (e.g. dog is a hyponym of canine since every dog is a canine)
    - Meronymy: part-whole relationship (e.g. paper is a meronym of book, since paper is a part of a book)

- Wikidata: Community project to create an open database of structured data.
  - Data curation model like WikiMedia
  - Intended to support Wikipedia (InfoBoxes)
  - 84 million entities (2020)
  - Multi-lingual
  - Both API access and full databases dump (JSON, RDF)
- Google knowledge graphs: Google's internal knowledge base to support the search engine.
  - Populated from FreeBase and internal data
  - Enriched with the support of schema.org
  - Accessible through API
- Schema.org: Collaborative community activity to create, maintain, and promote schemas for structured data on the Internet.
  - Sponsoring companies: Google, Microsoft, Yahoo and Yandex
  - Two types hierarchies: textual property values, things that they describe
  - Core vocabulary currently consists of 642 Types, 992 Properties and 219 enumeration values
  - Used by other knowledge bases (e.g. google knowledge graph API, DBpedia, etc.)
  - Different encodings can be used (i.e. JSON, RDFa (microformat for embedding RDF into HTML), Microdata)
- Linked Open Data: Repository of open data and knowledge bases and tools.

*Note: Only Schema.org is not an (instance-level) ontology.*

## 2.7 LEC12 : Information Extraction

The idea is to produce knowledge bases automatically by extracting knowledge from documents. The challenge is that knowledge is encoded in natural language. The objectives are:

- Automated or accelerated creation of knowledge bases
- Support for structured search on documents

For extracting knowledge (build a graph from documents) from textual content, we can consider the different constituents of a knowledge graph separately: entities, attributes and relationships. Basic question in knowledge *extraction*:

- Who are the entities?
  - Key phrase extraction: extraction of typical phrases in a document that could identify basic concepts.

  - Named entity recognition: extraction and typing of text that represents names of real-world entities.

- What are their attributes?
  - Named entity recognition

- How are they related?
  - Relation (information) extraction: automated extraction of relationships from text.
  - Taxonomy induction: automated extraction of a specific relationship, namely generalization, from text.

Once knowledge graphs have been extracted from text they can be further processed. This enables the inference of new knowledge from the existing knowledge, but as well the correction, completion and integration of existing knowledge bases. Basic question in knowledge *inference*:

- Who are the entities?
  - Entity linking/disambiguation: Identify which entity names represent the same real-world entity, respective which entity is referred to in case of ambiguous entity names.
  - Schema integration: Which classes, attributes and relationships in one knowledge bases correspond to which in another.

- What are their attributes?
  - Collective classification: The problem of learning unknown attribute values from the available knowledge in knowledge base.

- How are they related?
  - Link prediction: The problem of learning unknown relationships from the available knowledge in a knowledge base.

We will begin with knowledge extraction:

1. Key phrase extraction
2. Named entity recognition
3. Information extraction
4. Taxonomy induction

In lecture 14 we will see the knowledge inference:

1. Entity disambiguation
2. Label propagation
3. Link Prediction
4. Data Integration

| **EPFL** | is | located | in | **Lausanne** | in | **Switzerland** | , | next | to | **Lake** | **Geneva** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| I | O | O | O | I | O | I | O | O | O | I | I |
| ORG | | | | GEO | | GEO | | | | GEO | GEO |

Table 9: Example of NER, sequence of tags

### 2.7.1 Key phrase extraction

**Idea**: key phrase extraction is "the automatic selection of important and topical phrases from the body of a document". It is useful for document summarization, search, indexing, classification and opinion mining. It could be used for the creation of domain-specific thesaurus and taxonomy.

**Approach**: Generate candidate phrases and rank them.

- Candidate phrases
  - Remove stopwords
  - Use n-grams (generally in the range of 2 to 5)
  - Consider part-of-speech tags (POS)
- Base line ranking approach
  - rank candidate phrases of the document according to their tf-idf value
- Advanced approaches
  - Usage of many structural, syntactic features of the documents
  - Usage of external resources, such as Wikipedia, Wordnet

### 2.7.2 Named entity recognition (NER)

- **Task**: Find and classify names of people, organizations, places, brands, etc. that are mentioned in documents.
- '*NTLK NER*' and '*Spacy*' are two python libraries for NER. Here are some examples of the usage of NER
  - Named entities can be indexed, linked, etc.
  - Sentiment can be attributed to companies or products
  - Information extraction can use names entities as anchors
- We can see NER as a sequence of tags, indicating whether a word is inside ($I$) or outside of an entity ($O$). See Table 9 for an example. In fact this is a classification problem! Note that in this context also punctuation marks are considered as words, as they may carry important information on the presence of an entity.
- We can also see NER as a classification task with the following features, Neighbouring words and preceding labels. See Figure 58.
- Reusing the previous example (Table 9), the word "Lausanne" has the following features:
  - Word and neighboring words: Lausanne, in
  - Part-of-speech tags ($POS$): $POS(Lausanne) = NN$ (noun phrase)
  - Prefixes and suffixes: $prefix(Lausanne, 3) = Lau$
  - Word shape: $WS(Lausanne) = Xxxxxxx$

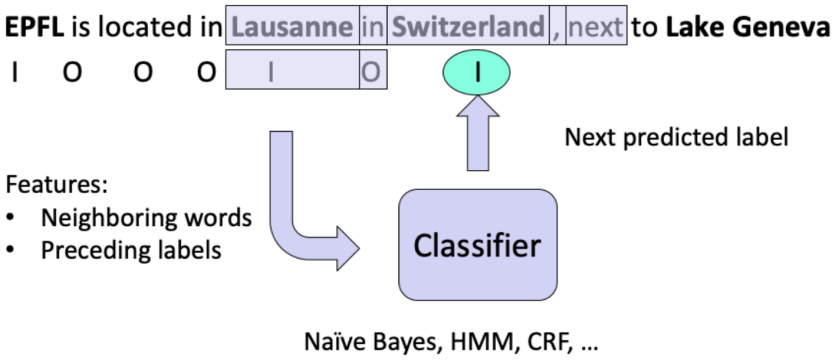


Figure 58: NER seen as classification task

  - Short wordshape: $SWS(Lausanne) = Xx$

- **Exploiting context**: When deciding the entity type exclusively on local context, important information may be missed. The idea is to consider a model that takes into account the sequential structure of language and exploits sentence context.

- **Generative probabilistic model**:

  - Sequence of words (known): $W = (w_1, w_2, \ldots, w_n)$
  - Sequence of labels (unknown): $E = (e_1, e_2, \ldots, e_n)$
  - Assume the text is produce by a probabilistic process: $P(E, W)$
  - Find the most probable model: $\text{argmax}_E\, P(E|W)$
  - Bayes' Law: $\text{argmax}_E\, P(E|W) = \text{argmax}_E\, P(E)P(W|E)$

  As we will not be able to estimate the complete probability distribution functions, we approximate them by making (sever) independence assumptions. We assume that the probability of a label to occur, depends only on the previous label. This corresponds to a bigram model.

  - Label **transition probabilities** (bigram model):

$$P(E) = P(e_1, \ldots, e_n) \approx \prod_{i=2,\ldots,n} P_E(e_i|e_{i-1})$$

  - Word **emission probabilities**:

$$P(W|E) \approx \prod_{i=1,\ldots,n} P_W(w_i|e_i)$$

- **Hidden Markov Model** (HMM): We can represent this model graphically using the HMM. See Figure 59. Learning the model: Maximum likelihood estimation (MLE) requires only counting (e.g. $P_E(I|O) = \frac{2}{4}$ and $P_W(in|O) = \frac{2}{5}$).

- **Smoothing**: Unseen words might only accidentally miss in the training data of length $n$: $P_{WS}(w_i|e_i) = \lambda P_W(w_i|e_i) + (1-\lambda)\frac{1}{n}$.

  For labels no smoothing is needed, as all labels occur in the training data.

- **Using the model**: For a given sequence of words $W$ find the most probable models for the labels $E$: $\text{argmax}_E\, P(E|W)$.

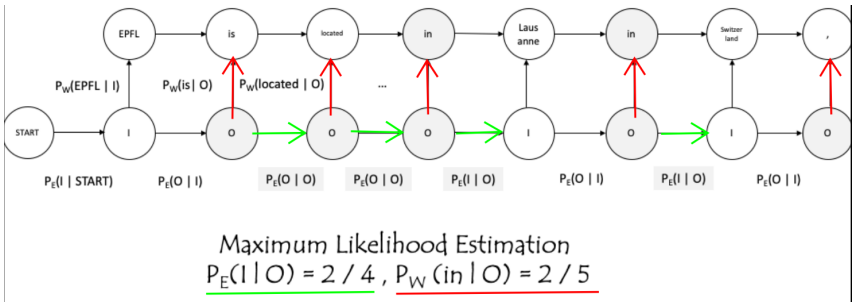


Figure 59: Example HMM

*Brute force search*: compute for all possible sequences $E = (e_1, \ldots, e_n)$ the probability $P(E|W)$ and then take the maximum. The complexity is $O(2^n) \implies$ unfeasible for longer sequences. However we observe that $\text{argmax}_E P(E|W)$ is independent of the choice of $e_n$ (last label) (see lecture 12 slide 26).

- **Viterbi algorithm**: Let $\pi(k, v)$ be the maximum probability a sequence of length $k$ can achieve with last label $v$. Then

$$\pi(k, v) = \max_u \pi(k-1, u) P_E(e_k|u) P_W(w_k|v) = \pi(0, *) = 1$$

This is a dynamic programming algorithm.

*Note: An HMM model would NOT be an appropriate approach to identify word n-grams. Note: The Viterbi algorithm works because it is applied to an HMM model that makes an independence assumption on the word dependencies in sentences.*

### 2.7.3 Information extraction

**Task**: Extract statement from text and create knowledge graphs.

Let see some "sample statements" $\implies$ typed statements:

1. EPFL - IS-A - Swiss Federal Institute of Technology $\implies$ ORG - PART-OF - ORG
2. EPFL - RELATED-TO - ETHZ $\implies$ ORG - RELATED-TO - ORG
3. EPF Domain - DEPENDS-ON - FDEA $\implies$ ORG - PART-OF - ORG
4. EPFL - LOCATED-IN - Lausanne $\implies$ ORG - LOCATED-IN - LOC

Statement extraction can be a tricky task due to the ambiguity and complexity of human language. Not every type of entity can be related in a meaningful way to another type of entity. For example, it would not make sense to have a statement expressing that a location is part of a person. For example we could have something like Figure 60.

There are multiples approaches to information extraction and we will discuss them in this lecture.

1. **Hand-written patterns**: A first approach to statement extraction is based on the observation that in natural language often a relationship is expressed in a regular fashion. Hearst was one of the first attempts to use regular expressions patterns to extract specific relationships (e.g. IS-A). The method is still used today.

   Hearst can be extended to different patterns than IS-A. In this approach we see that certain relationships can only hold among certain types of entities. Thus in a first step a named entity recognition is performed and then the patterns are searched for which matching

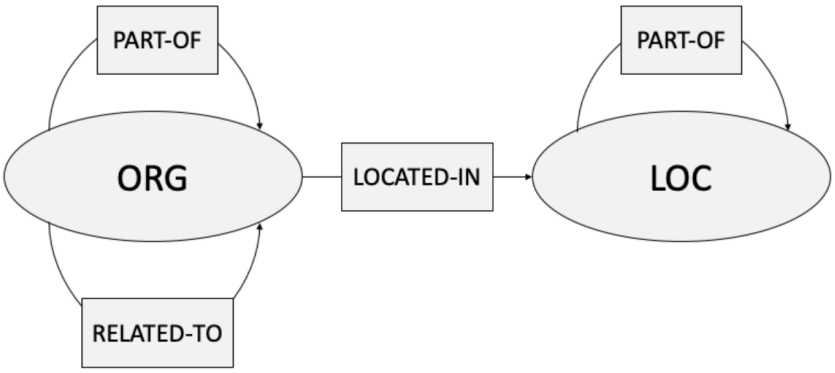


Figure 60: Statement schema

types of entities can be found (e.g. EPFL is located in Lausanne $\implies$ ORG is located in LOC).

Pros and cons of hand-written patterns:

+ Rules tend to be *high-precision*
+ Can be tailored to specific domains
- Human patterns are often *low-recall*
- A lot of effort to think of all possible patterns (even for experts)

2. **Supervised machine learning**: The approach requires first a training set (labeled data) to train a classifier on. However producing such a training set is labor-extensive. To create a training set:

   (a) Choose relevant named entities and their relations
   (b) Hand-label relations among entities (positive examples)

   Two-step approach:

   (a) A **filtering classifier** (e.g. Naïve Bayes), to detect whether a relation exists among the entities
   (b) A **relation-specific classifier** detecting the relation label

   To train the classifiers:

   - Extract named entities in the document corpus using NER
   - Detect pairs of entities, e.g. in the same sentence
   - Use unlabeled entity pairs as negative examples.

   The features being used for information extraction are typically different from the ones used for NER. When we have identified two occurrences of named entities in a sentence, also called two mentions (M1, M2), then we can identify the following features related to the two mentions:

   - The bag of words and bigrams found within the whole sentence
   - Distinct from that the BOW and bigrams in between the mentions
   - The headwords (these are words that are typically found in a standard dictionary)
   - Words in specific position with respect to the mentions
   - Stemmed versions of all the words above

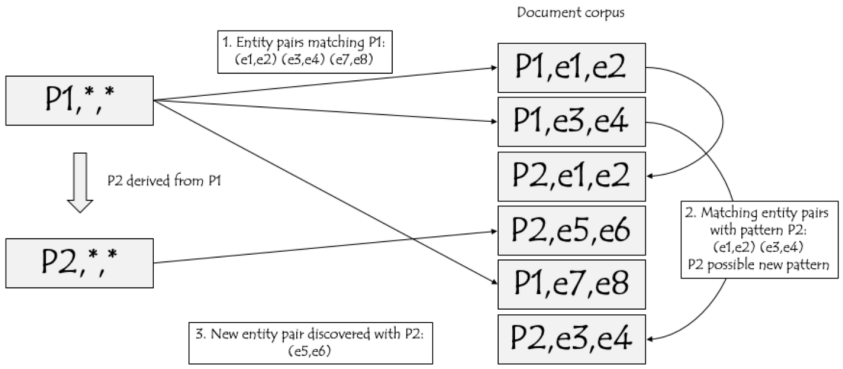


Figure 61: Example of Bootstrapping

- The type of entities used
- Syntactic features, extracted with part-of-speech analysis

Lecture 12 slide 43 talks about syntactic features. I did not understand that part.

Properties:

- Low precision
- High recall
- work intensive

*Note: Hand-written patterns are in general more precise than classifiers.*

3. **Bootstrapping**: No training data, but a few high-precision patterns. The basic idea is that one may have a few high-precision patterns, such as the Hearst patterns, or simple a set of example statements that are manually extracted, and one tries to generalize these patterns by analyzing a large document corpus. Approach:

   - Find entity pairs that match the pattern.
   - Find sentences containing those entity pairs
   - Generalize the entities in those sentences
   - Generate new patterns

   See Figure 61 for an example

   An other example:

   - Pattern: LOC is located in LOC
     - Mumbai is located in India
     - Adelaide is located in Southern Australia
     - Sriharikota is located in Nellore
   - Search for entity pairs (Mumbai, India)
     - Mumbai is India's top destination
     - Mumbai hotels, India
   - New patterns
     - LOC is LOC's top destination
     - LOC hotels, LOC

| Entities | | Text features (e.g. pattern) | | | Relation from knowledge base | |
|---|---|---|---|---|---|---|
| Entity 1 | Entity 2 | PER was born in LOC | PER was born to PER | PER and PER | Birthplace (X,Y) | Married (X,Y) |
| John Lennon | Liverpool | X | | | ? | |
| John Lennon | Julia Lennon | | X | | | |
| John Lennon | Julia Lennon | | X | | | |
| Julia Lennon | Alfred Lennon | | | X | | ? |
| Barack Obama | Hawaii | X | | | X | |
| Barack Obama | Michelle Obama | | | X | | X |

Table 10: Example of distant supervision

The problem is **semantic drift**: For example 'LOC hotels, LOC' matches also '[...] Geneva hotels, Lausanne hotels [...]' $\overset{?}{\rightarrow}$ Geneva is located in Lausanne?

**Confidence**: Assume we have a confirmed set of pairs of mentions $M$. A new pattern should also match many of those.

$$Hits_p = \text{number of pairs in } M \text{ that a new pattern matches}$$
$$Finds_p = \text{total number of pairs that a new pattern matches}$$

Confidence that a new pattern finds many relevant mentions:

$$Conf(p) = \frac{Hits_p}{Finds_p} \log(Finds_p)$$

If the new pattern retrieves more entity pairs that are already confirmed to be correct, its confidence increases. However, if in proportion it matches too many pairs, without creating confirmed entity pairs, the confidence in the pattern is lowered.

4. **Distant supervision** How to deal with the problem of lacking training data?

   **Idea**: Use existing knowledge bases to collect training data for building a classifier. Combines advantages of bootstrapping with supervised learning.

   Using entity extraction, entity mentions in text can be linked to corresponding mentions in a knowledge base.

   See Table 10 for an example. Assume "Barack Obama was born in Hawaii. Barack and Michelle Obama [...]" is known and accepted (blue zone). Now imagine we encounter "John was born in Liverpool, to Julia and Alfred Lennon." we can fill the yellow part. Now from the blue and yellow part we can try to fill the green one.

   The approach can be seen in Figure 62.

   **Features**: In distant supervision the features are constructed in a different way than in standard information extraction using supervised learning, due to the fact that the number of training examples is in generally much higher. Instead of producing a feature vector from combining all individual features, each feature combination is considered as a separate feature, which results in a much larger feature space. As a result, those more complex features are much more precise, but have low recall. This is, however, compensated by the fact that the training set is much larger.

   - Use conjunctions of standard IE features as sentence features
     - Match only if all individual features match
     - High precision, but low recall features!
     - Feasible, since training set is large

   The reader can see an example in lecture 12 slide 54.

   *Note: Distant supervision can help to detect rules.*

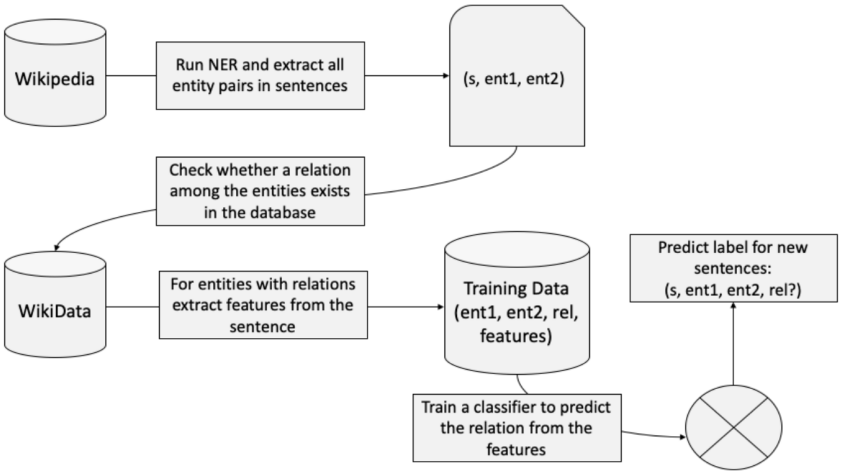


Figure 62: Approach of distant supervision

5. **Matrix Factorization**: Distant supervision aims at generating classifiers for relations that are based on syntactic features. Instead learning a classifier, create low-dimensional representations for entity pairs and relations. Using the same data as for distant supervision (entity pairs from text linked to relations from knowledge bases). We use those representations to

   - Link text patterns to relation types and identify similar text patterns
   - Extract relations from text

   **Matrix representation**: Create a matrix with

   - Entity pairs as rows
   - Relations types as columns
     - Relations from text patterns
     - Relations from knowledge base

   The entity-pair/relation matrix is a sparse matrix (like in recommender systems). See example lecture 12 slide 57.

   Using matrix factorization based on SGD might help to "guess" new relationships (as in recommender systems it helps to guess unseen ratings). This idea we will alter exploit also for the problem of link prediction. See Figure 63.

   **Bayesian personalized ranking**: The idea is to give observed true facts higher ranking than unobserved (true or false) facts. **Approach**: Create ranked pairs $f^+$ and $f^-$. Objective function:

   $$\sum_{f^+,f^-} \log \sigma(\theta_{f^+} - \theta_{f^-})$$

   where $\theta_f = p \cdot r$. Maximize with SGD.

   The matrix factorization obtained this way allows to map entity pairs and relationships in the same low-dimensional space.

## 2.8 LEC13 : Taxonomy Induction

### 2.8.1 Taxonomy induction

- Taxonomy induction aims at extracting related facts and organizing them in a structured knowledge basis, e.g. classification of animals.

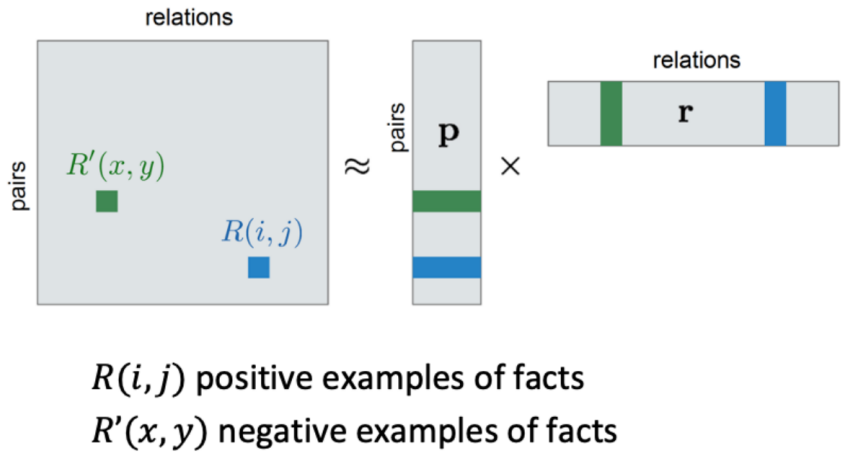


Figure 63: Matrix factorization

- One of the advantages of taxonomy induction (and more generally ontology induction) is the possibility to perform inferences on the extracted knowledge. Hyponyms can inherit properties from hypernyms (no need to learn extra facts separately). Due to transitivity of ISA, no need to learn inferred facts.
- *Note: The "correct" taxonomy depends on the perspective and application.*
- **Taxonomy induction task**: Starting from a root concept (e.g. animals) and a basic concept (e.g. lion).
  1. Learn relevant terms and their hypernym / hyponym relationships
  2. Filter out erroneous terms and relations
  3. Induce a taxonomy structure
- **Learning terms**: The basic idea is to learn terms and relationships by querying a Web search engine. The template approach (double-anchored patterns):
  - Given a root concept $c$ (e.g. animal) and a seed $s$ (e.g. lion)
  - Hyponym pattern: $P_i(c, s, X) = c$ such as $s$ and $X$ (detecting instances)
  - Hypernym pattern: $P_c(t_1, t_2, X) = X$ such as $t_1$ and $t_2$ (detection classes)
- **Finding hyponyms**: Recursively harvest new terms using a Web search engine.
  1. $T = \{s\}$
  2. $w(t) = 0$
  3. while $T$ changes: {
  4. for all $t$ in $T$: {
  5. submit $P_i(c, t, X)$ to search engine
  6. add to $T$ all new terms $t_{new}$ found in position $X$ in a result
  7. $w(t) + +$
  8. }}

  *Example: Search '"animal such as lion and"' on Google.*
- **Finding hypernyms**:

1. $C = \{c\}$
2. for all $t_1, t_2$ in $T$ with $w(t_i) > 0$:{
3. submit $P_c(t_1, t_2, X)$ to search engine
4. add new term $h$ found in position $X$ to $C$
5. add $t_1 ISA h$ and $t_2 ISA h$ to the hypernym relations $H$
6. $w(t_1, t_2, h) + +$
7. }

The filtering is simple:

1. rank concepts $h$ by $\sum_{t_1,t_2} w(t_1, t_2, h)$
2. keep top concepts

*Example: Search '"such as lion an tiger"' on Google.*

- **Inducing hypernym graph**: Many possible relationships among concepts and terms have likely not been discovered.

1. For each pair $t_1, t_2$ in $T \cup C$:{
2. Construct query $q_1 = h(t_1, t_2)$ and $q_2 = h(t_2, t_1)$
3. with Hearst pattern $h(X, Y)$, e.g. $h(X, Y) =$"$X$ such as $Y$"
4. Submit query to search engine and count the number of results
5. }
6. If $\#result(q_1) > \#result(q_2)$:{ add $t_1 ISA t_2$ to $H$ }
7. else: {add $t_2 ISA t_1$ to $H$}

The result is a directed hypernym graph.

*Example: Search '"lion such as animal"' and '"animal such as lion"' on Google*

- **Cleaning the hypernym graph**: After that, the graph $H$ may contain redundant paths (e.g. animal → chordates → vertebrates but also animal → vertebrates. Delete the shorter one). Please remove them.

1. Determine all **basic concepts** (not hypernym of another concept)
2. Determine all **root concepts** (have no hypernyms)
3. For each basic concepts - root concepts pair, select all hypernym paths that connect them
4. Choose the longest hypernym path for the final taxonomy

- We will present now a **more advanced technique** for automated taxonomy induction that we recently developed. It takes advantage of different advances in the available tools, but also incorporates some very fundamental ideas that have not been taken into account in the literature and largely help to improve the quality of results. The main ideas are the following:

  – Instead of starting from a clean predefined set of seed terms, the method starts from a document corpus and uses key phrase extraction to generate an initial vocabulary. This vocabulary may be noisy, but the method will help to clean it up while generating the taxonomy.

| | | |
|---|---|---|
| **TopEdge** | 2 | blintz→goody |
| | 3 | blintz→goody→thing |
| | 4 | blintz→goody→ulead→editor |
| | 5 | blintz→goody→ulead→social networking →networking→part |
| | 6 | blintz→goody→ulead→editor→storyliner→role |
| | 2 | oat→food |
| | 3 | oat→crop→thing |
| | 4 | oat→crop→total loss→partial loss→loss |
| | 5 | oat→cereal grain→grain→balanced diet→diet→factor |
| | 6 | oat→cereal→industry→field of life→other carrier→carrier |

Figure 64: Example of semantic drift

  - Instead of querying the Web, the method uses a Hypernym database that has been generating from analyzing a Web corpus. (WebIsADB)
  - Instead of performing simple statistics, the method uses various machine learning techniques for inducing the taxonomy.

- Current approaches in the literature make two main assumptions:

  1. The vocabulary is free of noise (Requires manual cleaning step before taxonomy induction)
  2. They assess taxonomy quality by estimating the probability of correctness of individual hypernym relationships (There is evidence that this works not well for more general terms)

  The second is important because the other relationships the terms have, are not considered and thus important information is lost. (**Semantic drift**)

- **Semantic drift in generalization**: Considering only individual hypernym relationships frequently results in semantic drift, in particular at the higher levels of the taxonomy. See Figure 64 for an example.

- Key ideas based on the two previous observations.

  1. Allow noisy vocabulary, but clean the resulting taxonomy after induction (After taxonomy induction more information is available to identify noisy terms)
  2. Estimate the probability of correctness of a complete path of hypernym relationships! (More contextual information is exploited in assessing the correctness of hypernym relationships)

- **Approach of no semantic drift method**: Find a DAG of generalizations.

  - Starting from one seed term in the vocabulary (e.g. apple)
  - Only one path for each hypernym of the seed term

  See Figure 65 for an example. How to construct a DAG? Let's talk about the probabilistic model.

- **Probabilistic model**:

  - *Ideally*: Given the evidence $E$, find the most probable graph $G$. $\text{argmax}_G P(G|E)$ (not feasible)
  - *Approximation*: $\text{argmax}_G P(G|E) = \text{argmax}_G(\prod_i P(E|S_i) \times P(S_i))$ where $S_i$ is a subsequence from seed term to a root.

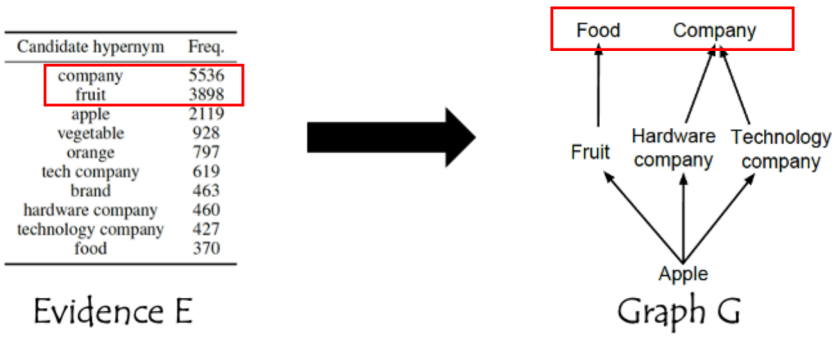


Figure 65: DAG example

- We still use an independence assumption but weaker than assuming all hypernym edges are independent of each other!
- We still need to estimate $P(E|S_i)$ and $P(S_i)$.

- **Solution for prob. model**:

1. **Estimation of probabilities of subsequences**
   - Estimate $P(S)$: For

$$S = t \to h_1 \to h_2 \to \ldots \to h_n$$
$$P(S) = P(t, h_1) \times P(h_1, h_2) \times \ldots \times P(h_{n-1}, h_n)$$

     where $P(a, b)$ is an edge probability. $P(a, b) \propto exp(w \cdot f(a, b))$. Edge features $f(a, b)$:
     * Normalized count, $n(a, b) = \frac{freq(a,b)}{\max_c freq(a,c)}$
     * Normalized difference
     * String-based features (prefix, suffix, substring, length)
     * Generality based features
     * Weights $w$ obtained from a classifier trained on manually annotated set of edges.
   - Estimate $P(E|S)$: For

$$S = t \to h_1 \to h_2 \to \ldots \to h_n$$
$$P(E|S) = \sum_j P(E_j|S)$$
$$P(E_j|S) \propto \max(sim(E_j, h_j)) \text{ , for all } h_i$$
$$sim(a, b) = \max(P(a, b), P(b, a))$$

   - Intuitive interpretation:
     * $P(S)$ promotes subsequences, which consists of individual edges with a larger probability of hypernymy.
     * $P(E|S)$ promotes subsequences, which contain a larger number of candidate hypernyms from $E$.
2. **Search strategy for subsequences**: Basic algorithm:
   (a) Iterates over all candidate hypernyms $h$ of $t$
   (b) For each $h$ perform a depth-limited beam search over the space of possible subsequences by recursively exploring the candidate hypernyms of $h$

(c) For each $h$ choose the subsequences $S$ with the highest score

(d) Choose the top-$b$ candidates based on sequence scores

The problem is to find best subsequence for term $t$, with first edge $t \to h$.

3. **Optimizing the resulting DAG**: Objective: prune the DAG composed of the set of subsequences constructed (noise removal). The flow network optimization:
   - demand controls how many seed terms should be discarded
   - Cost related to quality of paths

- *Note: the subsequence approach aims to avoid the semantic drift, thus its results are better than top edges seen at the beginning of lecture 13.*

## 2.9 LEC14 : Knowledge Inference

Basic question in knowledge inference:

- Who are the entities?
  - Entity linking/disambiguation
  - Data integration
- What are their attributes?
  - Collective classification
- How are they related?
  - Link prediction

### 2.9.1 Entity disambiguation

- **Task**: Link a text mention in a document to an entry in a knowledge base (e.g. WikiPedia or WikiData) (also called entity resolution and linking)
- **Challenge**: Two problems:
  - Homonyms: entities with the same name (e.g. *Schindler*'s list vs *Schindler* group)
  - Synonyms: different names for the same entity
- **Sources of information**:
  - **Local**: Similarity of a text mention of the entity and the data in the knowledge base (e.g. *Schindler* $\approx$ *Schindler's list*)
  - **Global**: Coherence of the different text mentions of potential entities within a document with respect to a knowledge base. Since these entities from the same text are likely to be in some form of relationship among each other, it is likely that such relationships are also discovered in the knowledge base. $\to$ entity graph
- **Entity graph**: (See Figure 66)
  - $m$ are text mentions of entities (extracted using NER)
  - For each mention there are candidates in KB ; these can be identified using local information from the text mention
  - Each is a graph node ; we can also associate a similarity measure to each node

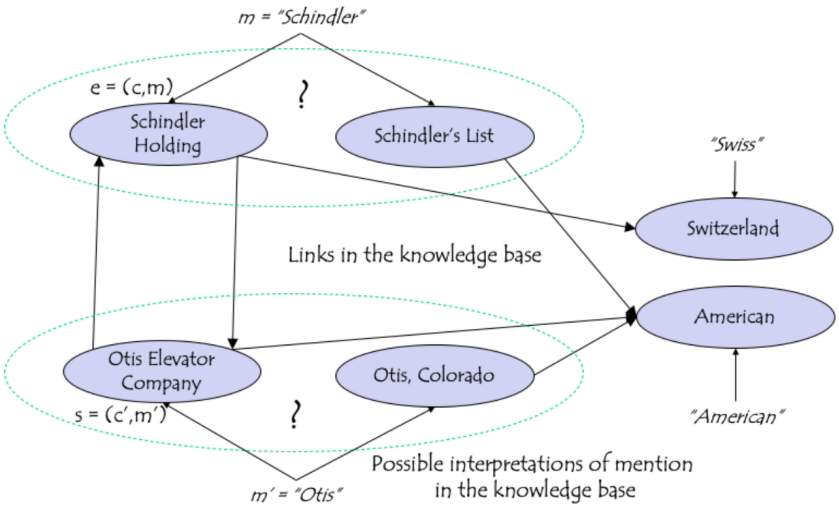


Figure 66: Example of an entity graph

  - Edges between mention-candidate pairs are included if a link exists in the KB among the respective candidates

- To perform entity disambiguation, we pose the following question: given a node $s$ in the entity graph, how well does it support the presence of another node $e$ in the graph? (See Figure 66 for $s$ and $e$).

  To answer this question a variation of the PageRank algorithm, which determines general relevance, has been proposed. It is called Personalized PageRank (PPR), which determines relevance with respect to a specific node in the graph.

- **Personalized PageRank** (PPR): Same as PageRank, except that random jumps are always back to the same node (or set of nodes). The original motivation was to be able to use bookmark list as source of rank. See Figure 67.

$$\vec{p} = c(qR \cdot \vec{p} + (1-q)\vec{e})$$
$$\vec{e} = (1, 0, \ldots, 0)$$

  Can be computed using Monte Carlo method:

  - Perform multiple independent random walks
  - Compute distribution of end points of random walks

- **Approach**: Finding the concept candidate linked to a mention $m$ that is most likely to be valid.

  1. For all concept candidates $c$ compute total support received from other nodes

$$e = (c, m)$$
$$s = (c', m')$$
$$score(e) = \sum_{s \in Contributors_e} PPR_s(e)$$

     See contributors nodes on next point

  2. Select the candidate with highest score

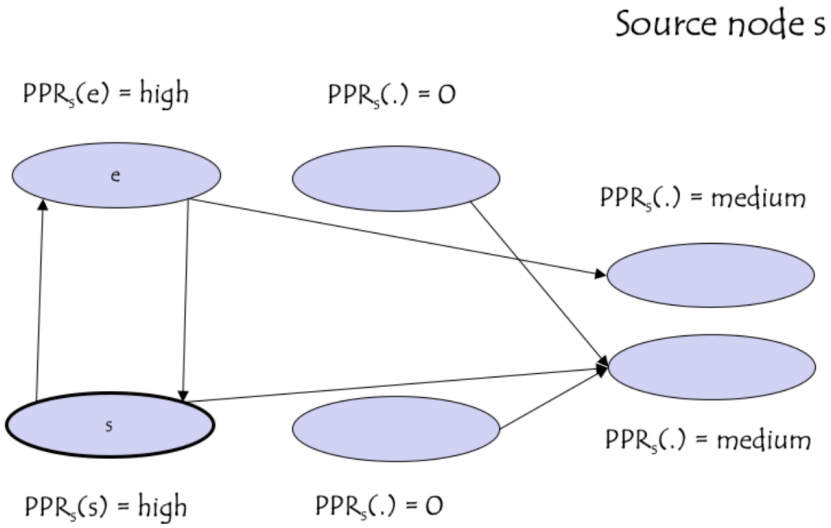


Figure 67: Example of PPR on entity graph

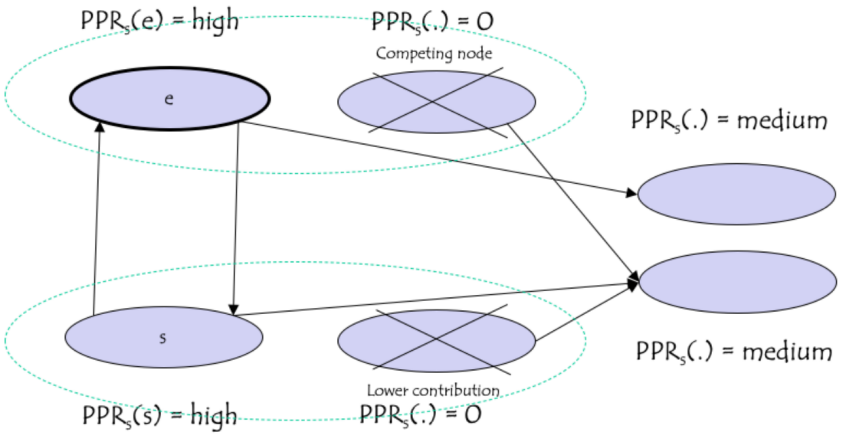


Figure 68: Contributor nodes

- Contributing nodes: *Only one* interpretation $c$ for a mention $m$ is valid.
  - Competing nodes $s = (c', m)$ that have the same entity mention as $e = (c, m)$ cannot support $e$
  - For multiple nodes $s$ that have the same entity mention $m$, only the one with the highest contribution is considered

$$Contributors_e = \{m',\ \operatorname*{argmax}_c PPR_{(c,m')}(e),\ m' \neq m\}$$

  See Figure 68 for an example.

Pas compris

- **Considering popularity**: The method can furthermore consider popularity measures for nodes (e.g. its degree). If information is insufficient, favor popular nodes.

$$score(e) = PPR_{avg} pop(e) + \sum_{s \in Contributors_e} PPR_s(e) pop(s)$$

  The first part promotes popular nodes while the second one promotes *contributions* from popular nodes.

- *Note: This is FALSE: Named entity recognition addresses the problem of synonyms.*

- *Note: Named entity recognition addresses the problem of entity classification.*
- *Note: Entity recognition addresses the problems of synonyms and entity classification.*
- *Note: Other concepts linked to the same mention cannot contribute to the score of a mention linked to a concept.*

### 2.9.2 Label propagation

- **Inferring attribute values**: Example problem: "Which users on Twitter have positive or negative emotion towards a topic?"
  - Users are nodes in a graph (follower network)
  - Emotion is an attribute of the node.

  A potential source of information in the case of Twitter is the emoticons in tweets. They indicate stance of user towards the topic. However only a (small) fraction of the users is using emoticons.
- **Propagating attribute values**: Assumption: nodes that are connected by an edge, have a higher propensity of sharing the attribute of interest (i.e. Twitter users following each other are more likely to share the same emotion towards a topic).
- **Model**: Graph $G = (V, E)$
  - Label set $L$ of size $n$
  - Vertices $V$ have a label from a set $L \cup \{unknown\}$
  - Edges are undirected and unweighted

  Objective:
  - Determine for a vertex $v$ a label vector $l_{inferred}(v)$ of size $n + 1$
  - $l_{inferred}(v)$ assigns a label probability

  To compute $l_{inferred}(v)$, we assume that all neighbors exert the same influence on a node. Thus we would require that:

$$l_{inferred}(v) = \frac{1}{\deg(v)} \sum_{(v,w)\in E} l_{inferred}(w)$$

  This is a recursive equation respectively a random walk model (like PageRank).
- **Adding pre-existing knowledge**: Initial knowledge on labels:
  - Known labels
    - $l_{apriori}(v)$ is a vector of size $n + 1$
    - assigns weight 1 for label of known for $v \in V$
  - Unknown labels
    - $l_{unknown}$ is a vector of size $n + 1$
    - assigns weight 1 for label *unknown*
- **Label propagation algorithm**:
  1. $l_{inferred}(v) = l_{apriori}(v)$ for nodes with known labels, otherwise $l_{unknown}$
  2. while not converged:{

3. $l_{inferred}(v) = \frac{1}{\deg(v)} \sum_{(v,w)\in E} l_{inferred}(w)$
4. $l_{inferred}(v) = p_v^{inj} l_{apriori}(v) + p_v^{con} l_{inferred}(v) + p_v^{aba} l_{unknown}$
5. }

Where the three parts of step 4 are:

1. For vertices with apriori labels, at every step the apriori distribution is injected with a certain probability $p_v^{inj}$ that depends on the vertex
2. The propagation of the label distribution to neighbors occurs then with a (remaining) probability of $p_v^{con}$
3. For all vertices the propagation process can also be abandoned with a certain probability $p_v^{aba}$

**How to determine these probabilities?** Entropy of transition probabilities: $H(v) = -\log \frac{1}{\deg(v)}$

$$
\begin{aligned}
c_v &= \frac{\log 2}{\log(2 + \deg(v))} \\
d_v &= (1 - c_v)\sqrt{H(v)} \text{ , if } v \text{ is labelled, 0 otherwise} \\
z_v &= \max(c_v + d_v, 1) \\
p_v^{inj} &= \frac{d_v}{z_v} \text{ , for labelled nodes, 0 otherwise} \\
p_v^{con} &= \frac{c_v}{z_v} \\
p_v^{aba} &= 1 - p_v^{con} - p_v^{inj}
\end{aligned}
$$

- **Behavior of probabilities: Labelled nodes**:
  - Injection probability increases with the degree of the nodes, while continuation probability decreases
  - Abandoning probability is positive only for very low degree nodes ($d \in 1, 2$ more or less)
- **Behavior of probabilities: Unlabelled nodes**: Abandon probability increases with degree. It prevents algorithm from propagating information through unlabeled, high-degree nodes.
- **Extensions**: Label propagation can be extended to
  - A priori knowledge given as probability distribution
  - Graphs with weighted edges
  - Directed graphs

  Alternative algorithm: MAD (modified absorption):
  1. Formulates an optimization problem and solves it directly
  2. Slightly better performance in practice
- **Discussion**: Label propagation is an example of a **semi-supervised learning** algorithm.
  - Exploit partial labelling

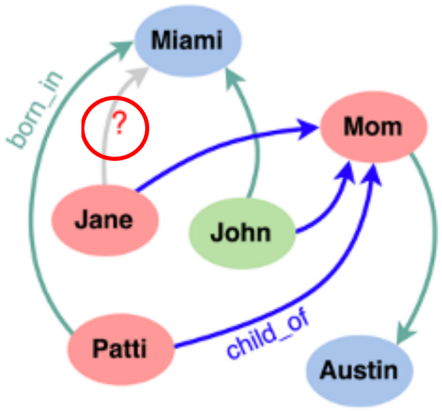


Figure 69: Example of knowledge base completion

  - Useful in cases where labels are sparse or labels can produced only for special cases using heuristics or background knowledge
  - Require that relationships among entities and their labels are correlated by some underlying principle

- *Note: Different neighbors of a node $v$ have exactly the same influence.*
- *Note: The probabilities $p_v^{inj}, p_v^{con}$ and $p_v^{aba}$ depend on both node degree and pre-existing knowledge.*

### 2.9.3 Link prediction

- Large knowledge bases are usually incomplete. Try to predict missing links from existing data. Look at Figure 69, Is Jane born in Miami? We already observed this problem for word embeddings (relationships can be encoded as linear transformations).
- **Model**: Knowledge graph $G$ consists of (correct) triples $(h, r, t)$ where $h$ (head),$t$ (tail) $\in E$ and $r \in R$. Define a plausibility score such that $f(h, r, t) > f(h', r', t')$ where if $(h, r, t)$ is a plausible triple and $(h', r', t')$ is an implausible triple.
- **Learning the model**: Minimize the loss function:

$$J(\theta) = \sum_{\substack{(h,r,t)\in G \\ (h',r',t')\in G'(h,r,t)}} \max(0, \gamma + f(h, r, t) - f(h', r', t'))$$

  where $G'(h, r, t)$ is a set of incorrect triples, generated by corrupting the correct triple $(h, r, t)$ and $\gamma$ is a hyperparameter. The minimization will be performed as usual using SGD.
- **TransE model**: This is one of the first embedding-based models for knowledge base completion. This model is based on the intuition from text word-embedding

$$f(h, r, t) = \|v_h + v_r - v_t\|_{1/2}$$

  where each entity and relationship is mapped to a low-dimensional vector, resulting in $v_h, v_r$ and $v_t$. It directly introduces a vector $v_r$ that represents the linear transformation corresponding to a relationship, and mapping the head and tail into the same (low-dimensional) vector space. The mappings from the entities and relationships to the latent space are performed (as in word embeddings) using embedding matrices, which constitute the model parameters that have to be learnt using SGD.

- **Performing SGD**:
  1. Initialize vectors with random values
  2. From interval $[-\frac{6}{\sqrt{k}}, \frac{6}{\sqrt{k}}]$ where $k$ is the embedding dimension
  3. In each iteration:
     (a) Sample a correct triple or batch (several triples)
     (b) Derive a corrupt triple from the correct one: replace $h$ or $t$ (not both) by a random entity
     (c) Update embeddings by minimizing loss function
     (d) Normalize all entity vectors to 1 (not relationship vectors!) (this avoids the model to find a trivial solution)
- *Note: The score function $f(h, r, t)$ is always positive and does NOT produce a vector.*

#### 2.9.4 Data integration (not in exam)

lecture 14 slide 44

FINISHED!