Summary Information Retrieval

Moritz Hoffmann

February 4, 2011

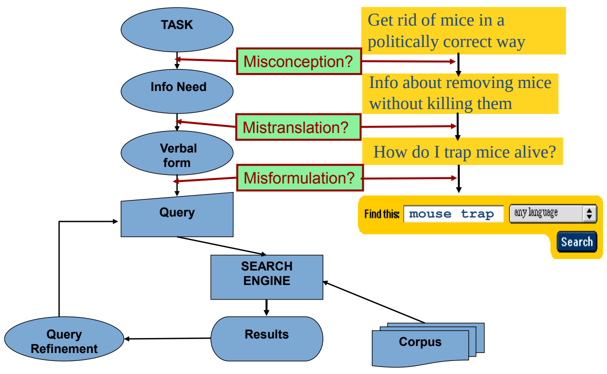


Figure 1: The classic search model.

1 Introduction

Information Retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

Data can be structured (relational data bases), semi-structured (XML) and unstructured as in free text search.

Basic assumptions for Information Retrieval Systems:

- Collection: fixed size set of documents.
- Goal: Retrieve documents with information that is relevant to the user to complete a task.

Quality constraints for an Information Retrieval System:

- Precision: Fraction of the retrieved documents that are relevant to the user.
- Recall: Fraction of relevant documents in collection, which are recalled.

Precision and recall is always a trade off, it is easy to set one of them to 1 and the other to 0, but being good at both is difficult. It depends on the use case whether to optimize for precision or recall.

1.1 Index

To store an index, one needs a mapping from words to texts the words occur in.

Term-document matrix Store words and documents as matrix columns and rows. Put a 1 when a word occurs in a document, otherwise a 0. Leads to very sparse data. Can answer Boolean queries using the matrix.

Incidence vectors For each word, store a vector which document contains this word. If document n contains the word, put a 1 at the n^{th} position in the vector. Can use bit operators to compute a query's result.

It is not possible to store an incidence matrix as it requires too much space. Better use an inverted index. For each word of the dictionary, store the documents they contain it (postings). Each document is identified by an ID.

$\boxed{\text{Word}_i} \rightarrow \boxed{\text{Document}_1} \ldots \boxed{\text{Document}_n}$

The postings can be a linked list of document IDs, which are sorted. For each word, the frequency can also be stored.

Construction of an inverted index:

1. Tokenize: Split the input documents into separate words. Remove punctuation.
2. Linguistic analysis: correct case, remove plurals etc.
3. Indexer: insert words into inverted index.

To search an inverted index, the postings of the terms need to be merged. The running time is in $\mathcal{O}(n+m)$, for n and m being the posting list's length. Here, it is required that the postings are sorted. It is also possible to do an N-way merge.

1.2 Discussion of Boolean Search

- No ranked results.
- Term frequency not relevant to result.
- No phrase search.

1.3 Terms

Clustering Given a set of documents, group them into clusters based on their contents.

Classification Given a set of topics, plus a new document, decide what topics it belongs to.

Ranking Learn how to best order a set of documents, i.e the result of a search.

1.4 Sophisticated Information Retrieval

- Cross-language retrieval.
- Question answering.
- Summarization.
- Text mining.

2 Building the Index

The following steps are applied on documents when building an index, and on a query when searching the index.

2.1 Tokenization

The input consists of documents, the output are tokens. Tokens are sequences of characters, which may end up in the index after further filtering. During this phase, punctuation is removed. Difficulties arise from terms that contain punctuation ('s, -) or consist of several words (San Francisco). Numbers are problematic as well, as they often contain punctuation. Older IR systems did not index numbers. In some language, there are compound words that should end up in several tokens (especially in German). Can use a compound splitter to avoid problem. Other languages have no spaces at all (Japanese or Chinese).

2.2 Stop words

Remove common words from the index. The idea is that words without semantic meaning do not need to be put into the index. Can save a lot of space, but reduces quality of index. Today, it is often compensated by good optimization.

2.3 Normalization

Match different forms of a word to one common form, like U.S.A to USA. Remove accents, umlauts. Care must be taken to only work on the source language: German MIT $\neq$ English MIT. Case folding to get one common spelling.

As an alternative, it is possible to add all expansions to a search in order to find all different spellings and cases for a term. This can be more powerful, but less efficient.

2.4 Thesauri and Soundex

A thesaurus can be used to add words to a search query the user has not entered. This way, synonyms and homonyms are handled automatically. Soundex can be used to search for terms that "sound" similarly.

2.5 Lemmatization

Lemmatization deals with reducing inflection and variants to one base form (am, are, is $\rightarrow$ be), it reduces words to their form in a dictionary.

2.6 Stemming

Reduce words to their stem by cutting off ends that contain no or little information. It is a crude and language dependent way to decrease the number of terms in the index. The best known algorithm for English is Porter's algorithm. The algorithm has a set of rules to shorten suffixes while in every iteration the longest suffix is shortened. This process is repeated a limited number of times (about 5 times).

Stemming is language dependent and thus introduces problems. It reduces precision and enhances recall. Its usefulness depends on the language it is applied on.

2.7 Data Structures

To quickly get results from inverted indexes, special data structures are required.

A simple and yet effective approach is to use skip lists or points. In this data structure, every couple of nodes a special node with two outgoing edges is stored. The additional edge points to an element further down the list. A simple heuristic is to use $\sqrt{n}$ skip pointers for a list with n elements. Modifications are problematic for this data structure. Today, the additional memory requirement may outweigh the faster access.

2.8 Phrase Queries

A phrase query asks for terms occurring in a special order. For this, it is not enough to store posting lists.

Biword index Index consecutive pairs of words instead of single words. This allows two-word phrase queries, but fails partly for longer queries as it can produce false positives. Also, the index gets much bigger as all word pairs end up in the index. It is not possible to answer proximity queries with biword indexes. They are not commonly used.

Positional index Adapt posting lists to not only store the document a term occurs in, but also store the positions of individual occurrences. The index can be queried by an extended merging schema. It is also possible to answer proximity queries with it.

For English, the index tends to be 2 to 4 times as large as the non positional index, its size is about 35% to 50% of the input's size.

3 Dictionary structures

The dictionary consists of tokens, which map to a set of documents. Due to the size of the index, it is required to keep the dictionary small and efficient. We have several options to store the terms and to store the postings.

3.1 Term structure

The terms can be stored in a list or array structure, in a tree or as hashed values.

Hashes Instead of storing the whole term in the dictionary, store only a hashed equivalent of a term. This leads to faster look up time, but it is not possible to search for minor differences or to do a prefix search.

Tree Store terms in binary or B-tree. As there exists a standard ordering for terms, we can use trees. Using trees, we can answer prefix queries. The downside is that queries are slower (order $\mathcal{O}(\log N)$ if balanced) and that restructuring is expensive.

4 Wild-card Queries

Wild card queries allow to search the index for terms with an undetermined component. For example, the query `mon*` would match all terms starting with `mon`. With trees, this is equivalent to the range search of m: `mon` $\leq m <$ `moo`. For suffix searches, an additional tree with terms stored backwards is required.

For queries with a wild card in the middle `co*tion`, a merge of the results of the two queries `co*` and `*tion` would be necessary.

4.1 Permuterm

The permuterm index can answer more complex wild-card queries efficiently. For each word in the index, store its permutations according to the following pattern:

Word hello

Permutation hello$, ello$h, llohe, lohel, o$hell, $hello.

Query	Look up	Query	Look up
X	X$	X*	X*
*X	X$*	*X*	X*
X*Y	Y$X*		
X*Y*Z	Y$X* merge with Z$*		

Note: merging two results has to work, as the permuterm index refers to terms, not documents!

Table 1: Permuterm query modification.

Permuterm indexes are about four times as big as normal indexes (in English).

4.2 Bigram indexes

A k-gram is a sequence of k characters. For the bigram (or k-gram) index, all k-grams are enumerated. Word boundaries are replaced by a special symbol. A second inverted index from bigram to dictionary term keeps the document association.

Input This is an example.

Bigram t, th, hi, is, s, i, is, s, a, an, n, $e, ex, xa, am, mp, pl, le, e$

For each bigram, a list of words is stored:
$\boxed{\text{th}} \rightarrow \boxed{\text{this}}, \boxed{\text{there}}, \ldots$, i.e. all words containing th.

To query the index, the query term has to be split in bigrams. To search for `mon*`, we search for `$m` AND `mo` AND `on`. The results are filtered for false-positives (e.g. `moon`) and the terms are then looked up in the term-document index.

This approach is fast and space-efficient. However, it is expensive to evaluate queries with many wild-cards.

5 Spelling correction

All documents have spelling errors, especially this created by OCR software. For OCR software, other errors have to be corrected than when a text is typed.

Two principal uses:

- Correct documents to have a clean dictionary.
- Correct queries

Two flavors:

- Correct isolated words for their own misspelling.
- Apply context-sensitive spelling correction.

Two strategies:

- Retrieve documents indexed by the correct spelling.
- Return several suggested alternative queries with the correct spelling.

For isolated word correction, a lexicon can be used. The lexicon is either a standard lexicon for a language (Webster's, Duden) or generated from the text corpus. The task for a lexicon is to find the term word given an input term.

5.1 Edit Distance

For two strings, compute the number of edits to transform one to the other. Operations are insert, delete, replace and optionally transpose. Can be solved by dynamic programming.

5.2 Weighted Edit Distance

Same as edit distance, only that weight of operation depends on character weight. Similar characters have a smaller edit distance than very different characters (compare `n` to `m` and `n` to `q`). Can also be solved by dynamic programming with a weight matrix as additional input.

5.3 Using Edit Distance

To find a closest word, first generate all words with the edit distance up to n and intersect this generated list with all known (and correct) words. Show remaining terms as suggestion to the user. Can be automated by picking best term.

Computing the edit distance is slow and expensive.

5.4 n-gram Overlap

First, enumerate all n-grams in the query, then retrieve all terms matching any of the n-grams. Use a threshold to limit the number of matching terms (like 2 of 3 n-grams must match).

5.4.1 Jaccard-Coefficient

The Jaccard index can be used to calculate the similarity of two sets of n-grams:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

It is 1 if both sets are equal and 0 when they are disjoint. Define a threshold to accept a match or not.

5.5 Context-sensitive Spell Correction

Hit based spelling correction: Generate a list of close words and search for each query with the word alternative. Return the alternative that gets a lot of hits.

Biword spelling correction: Break phrase query into biwords and look for biwords that only require one word to be corrected. Enumerate matches and rank them.

5.6 General Issues

Spelling correction is used to present alternative to the user. Heuristics have to be used to retrieve the best match. Generally, spelling correction is computationally expensive, but can be optimized a lot.

5.7 Soundex

Class of heuristics to expand queries into their phonetic equivalents. The algorithm reduces every word into a 4 character reduced form. The algorithm is not very useful for IR, although most data base vendors offer it.

6 Compression

Compression is required to save resources, mainly disk space and memory. Its premise is that decompressing data is faster than retrieving it from disk and cheaper than keeping it in memory.

6.1 Lossy or Lossless?

Lossless compression keeps all information, while lossy compression discards some. Many of the text processing steps are lossy, like stemming, case folding and stop words.

6.2 Vocabulary vs. Collection Size

To estimate the number of unique terms in a set of documents, we can apply *Heaps' law.* It states that the number of terms M depends on the number of tokens T in the collection and $30 \leq k \leq 100$, $b \approx 0.5$:

$$M = k \cdot T^b.$$

6.3 Collection Frequency — Zipf's law

The CF is the number of occurrences of a term in a collection of documents. *Zipf's law* states that the i^{th} most frequent term has a frequency proportional to i^{-1}. cf_i is the collection frequency of term i.

6.4 Compression on Dictionary

The dictionary is a crucial part of the inverted index, which is accessed very often. Thus it should be kept in memory.

First, we assume the dictionary is an array of fixed-width records, with the term, its frequency and a pointer to the posting list.

As a string Instead of storing the term in the record, store a pointer to a position in a long string with all terms appended to each other.

Blocking Store pointers to every k^{th} string. In the dictionary string, the word length needs to be stored.

Front encoding Sorted words have a long common prefix, thus only store differences for last $k-1$ elements in a block of k. Related to general string compression.

6.5 Compression on Posting Lists

Posting lists should also be small to retrieve hits from the IR system. A posting list is typically ten times larger than the dictionary.

Store offsets Instead of storing each document ID, store the offset to the next to get smaller numbers.

Variable length encoding The length of document IDs differs a lot when storing offsets, so it is desirable to store variable length records.

Variable Byte (VB) For each byte, dedicate one bit as continuation value. If the number $g \leq 127$, encode it and set $c = 1$, otherwise encode lower 7 bits and iterate on the higher bits. Instead of bytes, nibbles or other units can be used. VB is uniquely prefix decodable.

Unary code Represent n as n 1 and a 0.

Gamma code Store length of binary encoded number plus the binary number without its leading 1. Example: 13 becomes 1110 101.

For number G it requires $2 \cdot \lfloor \log G \rfloor + 1$ bits. Gamma code is uniquely prefix decodable. Seldom used to to machine constraints on word boundaries.

7 Ranking

7.1 Ranked Retrieval

Tries to overcome limitations imposed by Boolean queries, like too many or few results. In ranked retrieval, the IR system returns the top ranked documents with respect to a query. Queries are free text rather than expressions in some language (like Boolean queries). If the ranking algorithm works, large result sets are not a problem as only the top ranked results are returned.

To calculate the rank, assign a score $\in [0, 1]$ to each document in a result set and order it by the score. The score describes how well document and query match.

One-term queries: score 0 for documents not containing the term, and higher the more frequent the term is used. A simple measurement is the Jaccard-coefficient, but it has problems as it depends on document and query length and ignores frequencies.

Term Frequency The term frequency $tf_{t,d}$ of term t and document d is defined as the number of times that t occurs in d.

Log Term frequency Instead of using a proportional scale, use the logarithm of the weight:

$$tfw_{t,d} = \begin{cases} 1 + \log tf_{t,d} & tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$$

The score s for a document-query pair (d, q) is defined as follows:

$$s = \sum_{t \in q \cap d} 1 + \log tf_{t,d}.$$

Document Frequency Idea: Rare terms are more informative than frequent terms. Let the document frequency df_t be the number of documents containing term t. N is the number of documents. The inverse document frequency idf_t then determines how seldom a term occurs:

$$idf_t = \log \frac{N}{df_t}.$$

td-idf Combine term frequency and document frequency. Measurement increases with occurrences of a term within a document and the rarity in the collection:

$$w_{t,d} = (1 + \log tf_{t,d}) \cdot \log \frac{N}{df_t}.$$

Score The score is the td-idf over all terms in the document-query pair (d, q):

$$\text{score}(q, d) = \sum_{t \in q \cap d} w_{t,d}$$

Document Vectors We can now represent each document by a vector of td-idf weights: $\mathcal{R}^{|V|}$.

Query Vectors We can either represent queries as vectors in the vector space or rank documents according to their proximity to the query space.

Vector space proximity Distance between two points. Euclidean distance is a bad choice as it is large for vectors of different length. Instead, use an angle. First, the vectors need to be length-normalized, then we can calculate the cosine between the two vectors:

$$\|\vec{x}\|_2 = \sqrt{\sum_i x_i^2}$$

and

$$\cos(\vec{q}, \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{|\vec{q}| \cdot |\vec{d}|}.$$

If the vectors are length-normalized, the cosine is simply the dot product:

$$\cos(\vec{q}, \vec{d}) = \vec{q} \cdot \vec{d} = \sum_{i=1}^{|V|} q_i \cdot d_i.$$

For vector space ranking, represent each document and query as a tf-idf vector. Then compute the cosine similarity score for the query vector and each document vector. Rank the documents according to the score and return the top k documents to the user.

8 Augmentation

8.1 Relevance feedback

After showing the results to the user, the user selects documents as relevant or not. The system uses this information to compute a better result. Idea: help the user formulating better queries.

8.2 Centroids

A centroid $\vec{\mu}(C)$ is the center of mass of a set C of points:

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

8.3 Rocchio Algorithm

Seek the query $\vec{q}_{\text{opt}}$ that maximizes:

$$\vec{q}_{\text{opt}} = \arg\max \big[\cos(\vec{q}, \vec{\mu}(C_r) \\ - \cos(\vec{q}, \vec{\mu}(C_{nr}) \big].$$

Problem: it is unknown what documents are truly relevant. If we know what documents are relevant, we can use a variant of Rocchio's algorithm:

$$\vec{q}_m = \alpha \vec{q}_0 \\ + \beta \frac{1}{|D_r|} \sum_{\vec{d}_j \in D_r} \vec{d}_j - \gamma \frac{1}{|D_{nr}|} \sum_{\vec{d}_j \in D_{nr}} \vec{d}_j.$$

D_r and D_{nr} are the sets of relevant and non-relevant documents, $\vec{q}_m$ is the modified query vector, $\vec{q}_0$ the original, and α, β, γ are hand chosen weights. $\vec{q}_m$ converges to n optimal solutions. For a high number of judged documents, increase β/γ. Negative weights have to be set to 0, positive feedback is more valuable than negative (set $\gamma < \beta$). Many systems do not allow negative feedback.

8.3.1 Assumptions

1. User has sufficient knowledge for initial query. Problems include misspellings, search language, vocabulary mismatch.
2. Relevance prototypes are "well-behaved". Requires term distribution in relevant documents to be similar and term distribution in non-relevant documents to be different from relevant documents. Problems include different names of the same thing, contradicting government policies.

8.4 Problems with Relevance Feedback

Long queries slow down the search process, which leads to long response times, thus higher costs. A solution is to limit the number of terms in a query. Users are reluctant to provide explicit feedback. Relevance feedback sometimes makes search results hard to understand (Why was this document returned?)

Relevance feedback is offered by some search engines, but hardly used by users.

8.5 Query Expansion

Idea: Add additional input on words or phrases. This can be done by adding terms from a thesaurus or by analyzing the result set to derive more terms. By adding terms to a query, we risk a significant decrease of precision. Also, maintaining a thesaurus manually is very costly.

8.5.1 Automatic Thesaurus Generation

A thesaurus lists words that are similar to each other. There are two general approaches to generate a thesaurus:

- Two words are similar if they co-occur with similar words. (more robust)
- Two words are similar if they occur in a given grammatical relation with the same words. (more accurate)

9 Evaluating Search Engines

- How fast does it index? Documents per hour, with average size.
- How fast does it search? Latency as a function of index size.
- Expressiveness of query language.
- User interface.

9.1 Precision and Recall

For a set of documents and a query, there are relevant and non-relevant documents. Some of the relevant are retrieved, and some of the non-relevant as well. See table 2 for more information.

Precision Fraction of retrieved documents that are relevant, i.e.

$$\Pr[\text{relevant}|\text{retrieved}] = \text{TP}/(\text{TP}+\text{FP}).$$

Precision typically decreases when returning more documents or when recall increases.

Recall Fraction of relevant documents that are retrieved, i.e.

$$\Pr[\text{retrieved}|\text{relevant}] = \text{TP}/(\text{TP}+\text{FN}).$$

Recall is 1 when returning all documents, and it is non-decreasing in the number of documents returned.

Accuracy Fraction of classifications that are correct. Not commonly used in IR.

$$(\text{TP}+\text{TN})/(\text{TP}+\text{FP}+\text{FN}+\text{TN})$$

	Relevant	Non-relevant
Retrieved	TP	FP
Not retrieved	FN	TN

Table 2: Precision and Recall

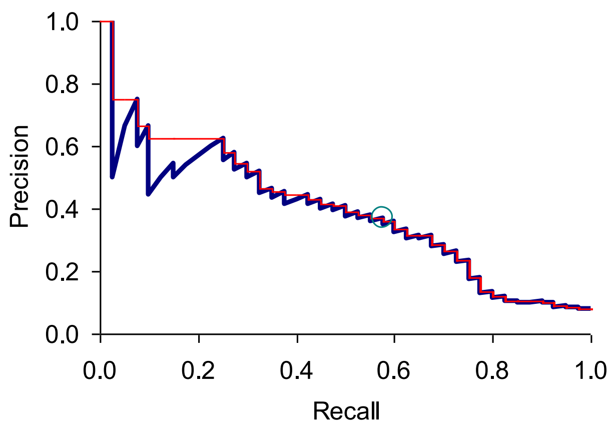


Figure 2: Precision-recall curve

9.1.1 F Measure

The F measure is the weighted harmonic mean. It takes both precision and recall into account:

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} = \frac{(\beta^2+1)PR}{\beta^2 P + R}$$

Typically, the F_1 measure is used, which is obtained by settings $\beta = 1$ or $\alpha = 1/2$.

9.2 Difficulties

It is hard to test the precision and recall of a search engine. Testing requires a test collection, with human classification. IR systems have problems with dealing with collections from different authors. Also, relevance is not the only measure for the quality of a search engine. The problems include duplicates and redundant information. Marginal relevance is a better measure of utility for the user. Use A/B testing to test the effectiveness of new features by redirect a small amount of users (1%) to a new engine and compare click through results.

To evaluate a search engine, draw a precision-recall curve. See figure 2 for an example.

10 Classification

For a description of an instance $d \in X$ and a fixed set of classes C determine the category of a document d: $\gamma(d) \in C$, where $\gamma(d)$ is a *classification function* whose domain is X and whose range is C.

For supervised classification, a training set D of labeled documents with each document $(d, c) \in X \times C$, determine a learning method for a learning classifier $\gamma : X \to C$.

10.1 Classification Methods

Manual Classify documents manually. Very efficient if done by experts, but does not scale. Very good for small problems and small teams.

Automatic Classification based on hand-coded rules. High accuracy reached for good filters and if filters are maintained by experts. Building and maintaining rules is expensive.

Supervised Machine learning. Examples are k-nearest Neighbors, Naïve Bayes and Support-Vector Machines. Requires hand-classified training data.

10.2 Probabilistic relevance feedback

Let t_k be a term, D_r be the set of relevant documents, D_{rk} be the subset that contains t_k, D_{nr} the set of known irrelevant documents and D_{knr} the subset of irrelevant documents containing t_g. Then we can apply Bayes' theorem:

$$\begin{aligned} \Pr[t_k|R] &= \frac{|D_{rk}|}{|D_r|} \\ \Pr[t_k|NR] &= \frac{|D_{nrk}|}{|D_n r|} \end{aligned}$$

To be continued at IR 7 page 15.

10.3 Naïve Bayes

Classification based on prior weights of class and conditional parameters indicating the relative frequency of c_j (P are positions):

$$c_{NB} = \underset{c_j \in C}{\arg\max} \left[\log \Pr(c_j) + \sum_{i \in P} \log \Pr(x_i|c_j) \right]$$

10.4 Rocchio Classification

Again, we represent documents as vectors in a highly dimensional vector space. Documents of the same class form a contiguous region and documents from different ones should not overlap. We can use relevance feedback for text classification, as it decides whether a document is contained in a class or not. We use td-idf vectors. As a starting point we use centroids for pre-defined classes. The test is done using cosine similarity.

The prototype (*centroid*) can be calculated by summing up all members of a class. The vector's length is irrelevant for cosine similarity. Classification is based on similarity to class prototype.

10.5 k-nearest Neighbors, kNN

The task is to classify a document d into a class c. Define the k-neighborhood N as the k nearest neighbors of d. Count the number of documents i in N that belong to c and estimate $\Pr[c|d]$ as i/k. Choose the class $\arg\max_c \Pr[c|d]$.

kNN is close to optimal. Error rate is less than twice the Bayes error rate as both query and training data contribute to error. Using 1NN is subject to errors as one could find a single atypical example or noise in a category. Usually, one chooses odd k to avoid ties, like 3 or 5.

NN depends on a distance metric, one can use Euclidean distance, Hamming distance (number of different features) or cosine similarity.

Finding the nearest neighbors requires to scan through all documents, $\mathcal{O}(|D|)$. This can be optimized by only comparing against the training data of size B: $\mathcal{O}(B \cdot |V_t|)$ and $B \ll |D|$.

kNN scales well with a large number of documents, there is no feature selection and training necessary, it is very accurate, but classes can influence each other and it may be expensive at testing time. Also, scores can be hard to convert to probabilities.

Naïve Bayes has a *low variance* and *high bias.* kNN has a *high variance* and a *low bias.* One has to trade off variance≈capacity and bias.

10.6 Linear classifiers and classification

For a two class classification, we could define a surface in the hyper space to separate the two classes. The hyper plane can be calculated iteratively, but there exist many solutions.

A hyper plane is defined as

$$\sum_{i=1}^{M} w_i d_i = \theta.$$

For Rocchio, we use

$$\begin{aligned} w &= \vec{\mu}(c_1) - \vec{\mu}(c_2) \\ \theta &= 1/2 \cdot (|\vec{\mu}(c_1)|^2 - |\vec{\mu}(c_2)|^2). \end{aligned}$$

For two class Naïve Bayes, we compute the log odds for a class C:

$$\log \frac{\Pr[C|d]}{\Pr[\overline{C}|d]} = \log \frac{\Pr[C]}{\Pr[\overline{C}]} + \sum_{w \in d} \log \frac{\Pr[w|C]}{\Pr[w|\overline{C}]}.$$

If the log odd is greater 0, we decide that d is in C. Set the hyper plane to

$$\alpha + \sum_{w \in V} \beta_w \cdot n_w \quad \text{where } \alpha = \log \tfrac{\Pr[C]}{\Pr[\overline{C}]}$$
$$\beta_w = \log \frac{\Pr[w|C]}{\Pr[w|\overline{C}]} \quad n_w = \text{occ. of } w \text{ in } d$$

For nonlinear problems, a linear classifier will produce bad results, while kNN works very well (with enough training data).

10.7 More than two classes

Any-of Classes are independent, a document can belong to any number of class, the problem can be decomposed into n binary problems. Very common for documents.

To classify a document, test if it belongs to a class for each class and decide individually.

One-of Classes are mutually exclusive, each document belongs to exactly one class.

To classify a document, test against all classes and assign to the class with highest score, confidence or probability.

11 Clustering

Clustering is the process of grouping a set of objects into classes of similar objects. It is the most common form of unsupervised learning. Clustering increases recall, enables better navigation or user interfaces and speeds up vector space retrieval. Possible outputs are hierarchies or visualization as maps.

The recall is increased as it can be assumed that documents in one classes behave similarly w.r.t relevance. Therefore, cluster documents a priori and return documents from a cluster matching a query. For example, a query for `car` should return `automobile` as well. For navigation, similar clusters could be shown to ask more specific questions.

Issues for clustering include the representation of clusters. Clusters could be represented as documents in the vector space. For a corpus it is also unclear how many clusters there are. Too many clusters include trivial clusters, too big clusters contain no information.

For the similarity notion, a semantic similarity is desired. In practise, one uses term-statistical similarity. We use cosine similarity in a vector space. The similarity is defined by a distance measure, e.g. Euclidean distance.

11.1 Algorithms

Flat algorithms start with a (random) partition and refine it iteratively, e.g. K means clustering. Hierarchical algorithms can work bottom-up, agglomerative or top-down, divisive.

11.2 Hard- vs. soft clustering

Hard clustering means that each document belongs to one cluster while in soft clustering each document can belong to several clusters. Hard clustering is easier to do, while soft clustering often makes more sense.

11.3 Partitioning algorithms

Construct a partition of n documents into a set of K clusters. The input is a set of documents and the number K. Find a partition of K clusters that optimizes the chosen partitioning criterion. Can choose a global optimum or an effective heuristic method (K-means and K-medoids).

11.4 K-Means

Work on real-valued document vectors. Clusters are based on centroids of points in a cluster c:

$$\vec{\mu}(c) = \frac{1}{|c|} \sum_{\vec{x} \in c} \vec{x}.$$

Reassignment to clusters is based on the distance to the current cluster centroids.

Algorithm: Select K random documents $\{s_i, 0 < i < K\}$ as seeds. Iterate until clustering converges: For each document d_i, assign it to a cluster c_j, such that distance $\text{dist}(d_i, c_j)$ is minimal. Next, update the centroids for each cluster c_j: $s_i = \vec{\mu}(c_j)$.

Termination can be based on a fixed number of iterations, unchanged partitions or unchanged centroids. K-means is known to converge against a state where clusters do not change, possibly with a large number of iterations.

11.4.1 Convergence

Define the goodness measure of cluster k as the sum of distances from documents to the cluster's centroid

$$G_k = \sum_i (d_i - c_k)^2,$$

and the goodness as $G = \sum_k G_k$. The goodness decreases monotonically. Convergence heavily depends on the right choice of seeds.

The time complexity of K-means is based on

- Computing the distance between two vectors: $\mathcal{O}(M)$.
- Reassigning clusters: $\mathcal{O}(KN)$ distance computations, resulting in $\mathcal{O}(KNM)$.
- Computing centroids: $\mathcal{O}(NM)$.
- For I iterations: $\mathcal{O}(IKNM)$.

K-means requires to number of clusters before running, however it is not really possible to know that number upfront. Modelling cost for each cluster and benefit, we can compute the total value, which should be maximized. The benefit of a document can be computed as the cosine similarity of a cluster with its centroid. The total benefit is the sum of all benefits.

11.5 Hierarchical Clustering

The goal is to build a tree-based hierarchical taxonomy from a set of documents. We use a recursive application of a partitional clustering algorithm.

11.5.1 Hierarchical Agglomerative Clustering

Start with each document in a separate cluster. Then repeatedly join the closest pairs of clusters, until there is only one cluster. This forms a binary tree of clusters. We define several different ways of computing the *closest pair*:

Single link Combine the two clusters that are most cosine similar. For an unspecified similarity function sim and two clusters c_i and c_j, we have

$$\text{sim}(c_i, c_j) = \max_{x \in c_i,\, y \in c_j} \text{sim}(x, y).$$

After merging c_i and c_j, its similarity to c_k is

$$\text{sim}(c_i \cup c_j, c_k) = \max(\text{sim}(c_i, c_k), \text{sim}(c_j, c_k)).$$

The resulting clusters can have a long and thin form.

Complete link Similarity of furthest points, least cosine similar. Similar to single link, we have

$$\text{sim}(c_i, c_j) = \min_{x \in c_i,\, y \in c_j} \text{sim}(x, y).$$

After merging c_i and c_j, its similarity to c_k is

$$\text{sim}(c_i \cup c_j, c_k) = \min(\text{sim}(c_i, c_k), \text{sim}(c_j, c_k)).$$

Clusters are tighter and more spherical.

Centroid Cluster those centroids are most cosine similar.

Average link Average cosine between pairs of elements.

Computational Complexity In the first iteration, there are $\mathcal{O}(N^2)$ steps required. In each iteration, the most recently created cluster has to be compared to $N-2$ clusters. This results in $\mathcal{O}(N^3)$ or if done cleverly $\mathcal{O}(N^2 \log N)$.

Group Average The group average is the average similarity between all pairs within merged cluster:

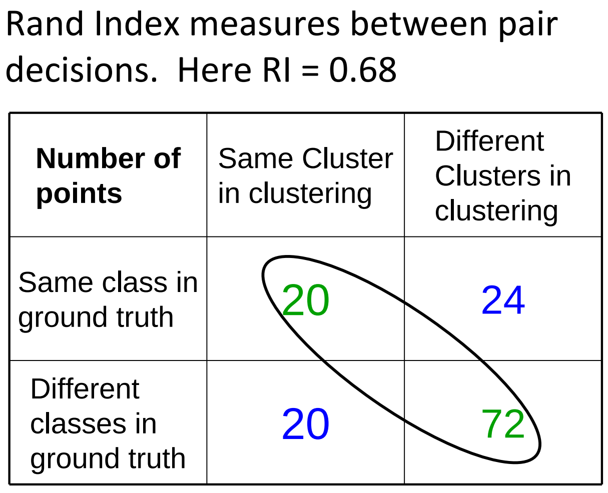
$$\text{sim}(c_i, c_j) = \frac{1}{|c_j \cup c_i| \cdot (|c_j \cup c_i| - 1)} \sum_{\vec{x} \in c_j \cup c_i} \sum_{\substack{\vec{y} \in c_j \cup c_i \\ \vec{x} \neq \vec{y}}} \text{sim}(\vec{x}, \vec{y}).$$

It is a compromise between single and complete link. One can choose to average across all ordered pairs in the merged cluster or to average over all pairs between the original cluster.

To compute the group average similarity, always maintain the sum of vectors in the cluster $\vec{s}(c_j) = \sum_{\vec{x} \in c_j} \vec{x}$ and compute the similarity in constant time:

$$\text{sim}(c_j, c_j) = \frac{(\vec{s}(c_i) + \vec{s}(c_j))^2 - |c_i| - |c_j|}{(|c_i| + |c_j|) \cdot (|c_i| + |c_j| - 1)}.$$

A good clustering makes sure that the intra-class similarity is high, and the inter-class similarity is low. The quality of a clustering algorithm is tested against a gold standard data. It must be assessed against a ground truth with labelled data. Assume document with C gold standard classes , while a clustering algorithm produces K clusters $\omega_1, \omega_2, \ldots, \omega_K$ with n_i members.



Rand Index measures between pair decisions. Here RI = 0.68

Number of points	Same Cluster in clustering	Different Clusters in clustering
Same class in ground truth	20	24
Different classes in ground truth	20	72

Figure 3: Rand Index schema

Purity The purity is the ratio between the dominant class in the cluster π_i and the size of the cluster ω_i:

$$\text{Purity}(\omega_i) = \frac{1}{n_i} \max_j(n_{ij}) \quad j \in C$$

This is a biased measure for n clusters.

Rand Index RI

$$RI = \frac{A + D}{A + B + C + D}$$

See figure 3 for a demonstration.

12 Latent Semantic Indexing (LSI)

Use the term-document matrix and convert it to a different representation using SVD.

For a term-document matrix A we compute an approximation A_k. Each term has its own row and each document its column. Thus, documents live in a space of $k \ll r$ dimensions.

In vector space, we can automatically select index terms and can handle partial matching. Ranking can be done according to a similarity score. To improve retrieval performance, one can apply term weighting schemas. Extensions include clustering and relevance feedback. It has a geometric foundation.

Lexical semantics cannot deal with polysemy (multiple meanings of one word) and synonymy (multiple terms with the same meaning).

12.1 LSI

The idea is that similar terms map to similar locations in the low dimensional space. Noise is reduced by the dimension reduction. LSI is clustering as related axes in the vector space are brought toghether.

- Perform a low-rank approximation of the document term matrix, typically rank 100-350. Under 200, it has been reported unsatisfactory. Lower k improves recall, higher improves precision.
- Map documents and terms to a low dimensional representation.
- Latent semantic space: a mapping such that the low-dimensional space reflects semantic associations.
- Compute document similarity based on inner product in the latent semantic space.

- Map each row and column of A onto the k-dimensional LSI space by SVD.
- Map a query q into this space by:

$$q_k = q^T U_K \Sigma_k^{-1}.$$

The dimensionality roughly shows the number of topics it presents. Mathematically, if A has a rank k approximation of low Frobenius error, then there are no more than k distinct topics in the corpus.

LSI can be applied to many pattern recognition and retrieval tasks with feature-object matrices.

13 Link Analysis

The Web can be viewed as a directed graph. One can assume that links have a signal quality, and the anchor text describes the link targets.

13.1 Page Rank

Idea: walk web pages by following links. At each stage, visit one of the links with equal probability. To avoid dead ends, do a random jump with 10% probability. This leads to a long term rate at which any page is visited. The rate can be computed by Markov chains.

Represent all pages in a matrix P. Then for all i, $\sum_{j=1}^{n} P_{ij} = 1$. Markov chains are ergodic, if there is a path from any state to any other state and if after a fixed time T_0, the probability of being in an any state in $T > T_0$ is nonzero. It converges to a long term visit rate. The state vector $a =$

$(a_i, \ldots, a_n$ is a vector of probabilities, where a_i is the probability to be in state i. For any state a, the next state is distributed as aP. If a is the steady state, we have $a = aP$, which is the left eigen vector of P. We can compute the steady state by $a = xP^k$ for some initial probability vector x.

For preprocessing, calculate the probability vector a from a given matrix. Each position represents the page rank of one page. For query processing, retrieve all matching documents and sort them by their page rank. This means that the order is query-independent. In reality, the page rank is used, but only within many other features. For page crawling policies, it is more often used.

The random surfer model used in the page rank algorithm can be biased, for example by bookmark pages. Also, the model does not really reflect a real surfer (no back button).

13.2 Topic Specific Page Rank

Page rank modified to return values based on query. There exist two approaches:

Off line Compute page rank for individual topics. Also query independent, but aware of categories.

On line Generate a dynamic page rank score based on weighted sum of topic specific page ranks.

To personalize a page rank query, we pass not only the web graph W, but also an influence vector v over the topics to the page rank algorithm. It outputs the rank wrt. the influence vector:

$$\sum_j [w_j \cdot \mathrm{PR}(W, v_j)] = \mathrm{PR}(W, \sum_j [w_j \cdot v_j])$$

13.3 Hyperlink-Induced Topic Search (HITS)

HITS is the predecessor of page rank. ...

14 Web Search

For web search, precision is much more important than recall. Precision is lowered by special pages prepared to be found by search engines with irrelevant content.

14.1 Size of the web

The size of the Web cannot be determined. However, it is possible to measure the size of a search index of a search engine. This is a rough measure of the web's size. The process can be automated by picking random queries from lexicon, and query two search engines with it. For each URL in the results of search engine A, check its presence in search engine B. The distribution induces the probability weight $W(p)$ for each page:

$$\frac{W(SE_A)}{W(SE_B)} = \frac{|SE_A|}{|SE_B|}.$$

For random searches, one can compare the result set size of the search engines. Then by averaging over all queries, using overlap and size ratio, estimate index size ratio and overlap.

By sending HTTP requests to IP addresses, one can estimate the size of the Internet. It has the benefit that it provides clean statistics and is not biased by a crawling strategy. However, many pages are not accessible like this (virtual hosts!), and not all pages are linked from the front page.

There is no perfect sampling method to solve the task, and although there are many new ideas the problem gets harder.

14.2 Duplicate Detection

Duplicates can be detected with finger prints. Near-duplicates require an approximate match, like edit distance or similarity measure.

A simple yet powerful method is to create word n-grams (shingles) from documents and then calculate the Jaccard coefficient to obtain the similarity of the two.