Introduction to Information Retrieval

2009

Christopher D. Manning

Prabhakar Raghavan

ch 1 – ch 21

444 pages

2016,11,17 - 2016,11,20

Information Retrieval (IR) is find material of an unstructured nature that satisfies an information need from within large collections.

broader than word → document term

A term-document incidence matrix helps in "boolean" retrieval → operators ∈ {AND, OR, NOT}

	doc1	doc2	doc3	doc4	...
term1	0	1	0	0	...
term2	0	0	0	1	...
term3	0	1	0	1	...

↳ We can answer queries by bitwise logical operations on rows for each term.

- effectiveness
  - precision: how much of what is returned is relevant?
  - recall: how much of what is relevant is recalled?

⇒ the incidence matrix is extremely sparse

term0 → doc0 → doc1 → ... (posting)
term1 → doc0 → doc1 → ...
⋮
termn → doc0 → doc1 → ... (posting list)

We call this data structure an **inverted index** which has the same principle as the adjacency list representation of a sparse graph

size of posting = document frequency

collect documents → tokenize text → normalize tokens → build index → usually sorted

collect documents: give each a unique id #

normalize tokens: ↳ linguistic processing

▶ To process boolean queries, we evaluate each conjunctive term in increasing document frequency, and calculate the intersection → linear merge. O(N)

• The extended boolean model also adds a "proximity" operator (# of intervening words, or some structural restriction, e.g. same sentence).

→ AND increases precision, decreases recall

→ OR increases recall, decreases precision

---

ch2

▶ We must decouple the actual encoding of the input from the system (UTF-8, ZIP, etc)

→ how granular are our documents? a recall/precision trade-off.

an instance of char sequence in the doc → token → type → term

type: the class of all tokens with the same char sequence

term: (normalized) form of token stored in IR system

- We process the queries with the same normalizers as we used with the documents.

~~Skip Others~~

- sometimes instead of trying to discover words, we might just use k-grams.

- "stop words" are words that can usually be ignored → sometimes have large impact on some queries.

• The general trend is to handle stop words more cleverly instead of ignoring them.

→ In normalization we often seek to form equivalence classes

→ accents and diacritics are often stripped

→ acronym normalization & case folding might have undesired side effects ⇒ C.A.T. ⇒ cat; Bush ⇒ bush

- normalization has many linguistically debatable issues.

⇝ Documents might contain phrases from many different languages.

Normalization

① Stemming: shorten the word according to some heuristic

② Lemmatization: perform linguistic preprocessing to find actual core (lemma)

	stemming	lemmatization
saw(v)	s	see
saw(n)	s	saw

→ neither offers hollistically great gains to modern IR systems.

- We can use skip pointers to speed up merging → great if indexes are static
  - ↳ $\sqrt{P}$ pointers for posting lists of size $|p|$

Positional Postings & Phrase Queries

① use biwords → xyz ⇒ {(x,y), (y,z)}
- ↳ avoid false positive by k-words
- ↳ extended biwords by part-of-speech tagging: head over heals ⇒ (head, heels)
- variable k: we have a phrase index
  - ↳ gets exponentially large

larger indexes, more complex intersections

② positional indexes: store each occurrence of the term in each document → this enables us to do proximity queries.

③ store positions but for common phrases and words with high document frequency store biwords

~26% more space, 75% speed-up

④ Keep a partial next-word index

---

ch3

looking up the terms requires a dictionary data structure
1. hashing: see CLRS(14)
2. binary trees: CLRS(13)
3. B-trees: see CLRS(27)

- hashing isn't great in the face an ever-growing vocabulary, doesn't let iterations
- balancing binary trees is expensive
  - ↳ B-trees can help

Wildcard Queries

generally makes retrieval slower

① Use ~~an~~ inverted B-Tree on top of the original, intersect the result and filter mismatches

② Keep permuterm indexes:

well → well$, ell$w, llwe, lwel → well → postings

and rotate query.

↳ prohibitively large

③ Pad with $ and do k-grams, intersect, and do post-filtering

Spell Correction

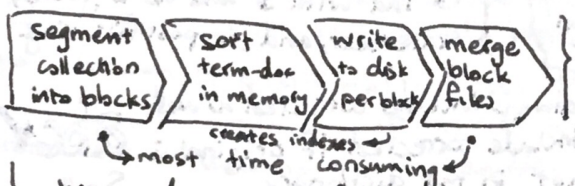
- always include corrected as well
- include corrected if original is not in the dictionary
- include corrected if result set is small ($< k$ for some k)
- suggest the correction to the user

- isolated → edit distance (Levenshtein)
  - ↳ use Jaccard coefficient $\frac{|A \cap B|}{|A \cup B|} < k$ to find candidates from corpus first
- context-sensitive → correct all words and query each result to find max
- phonetic correction: use soundex
  - herman, hermen → "H655"
  - ↳ uses equivalence classes for similar-sounding consonants

ch4

- The design of indexes is constrained by hardware
- We use caching to make up for disk speed
- Collocated data is faster to load
- OSs usually read data in blocks → we therefore use buffers
- CPU is free while reading ⇒ reading compressed data is faster

BLOCKED SORT-BASED INDEXING (BSBI)



↳ We need a mapping from term to term id

SINGLE-PASS IN-MEMORY INDEXING (SPIMI)

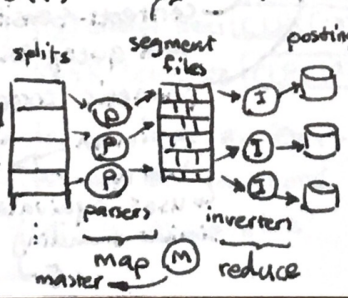
Instead of term ids, we create mappings from terms. We keep doing things in-memory, until memory is full, then write to disk.

↳ memory stuff & disk files can be zipped.

↳ no sorting ⇒ Θ(T)

Extremely large collections can be handled by distributed indexing via MapReduce

highly fault tolerant & scalable



Dynamic Indexing: store modifications in the memory, when it becomes too large, merge with disk. $\Theta(T^2/n)$ (T → total postings, n → memory index size)

Or: create indexes of increasing sizes & merge into that ($\Theta(T \log T_n)$) → put off merging disk blocks.

Usually we'd just reconstruct from scratch, as it is less complicated.

↳ Indexes are sorted based on docId so that we can merge.

⇒ For ranked retrieval, we might want to sort by some weight.

■ We usually handle authorization on retrieval time, since maintaining an ACL for changing permissions is difficult.

ch5

COMPRESSING the INDEX

① less disk space (duh!)
② better caching ← more things fits in RAM
③ faster transfer ← decompression overhead is worth it!

■ the 30 most common words account for 30% of written text ← rule of 30

▲ compression { lossy → when lost information is unlikely to be searched for
lossless

HEAP'S LAW: $M = kT^b$, $30 \le k \le 100$, $b \approx 0.5$
M: number of terms; T: number of tokens

ZIPF'S LAW: $cf_i \propto \frac{1}{i} \Rightarrow \log(cf_i) = \log c + k \log i$
cf_i: collection frequency of i^{th} most frequent term; c: well-chosen constant; $k = -1$

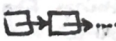
COMPRESSING THE DICTIONARY

① treat it as one large string and use pointers to end of words. Locate by binary searches ⇝ 60-30% decrease

② cut the string into k-term blocks
k ↘ decrease: better lookup, larger dictionary
k ↗ increase: slower lookup, smaller dictionary

③ Front coding: store common prefixes in the block and don't repeat it.

▶ the observation for compressing an index's postings come from the fact that the gaps in document IDs are smaller for more frequent terms & we might benefit by using variable lengths for posting list storage.



COMPRESSING THE POSTINGS

① Variable Byte encoding: first bit:
if 0, next byte is part of the posting
if 1, this is the last byte
⇝ 7-bit payload
↳ very effective, easy to implement

② γ code: 1110 101
unary representation of offset length ↳ offset with leading '1' removed

▶ It is universal: within a factor of the possible optimal, always

---

ch 6

→ fields: author, date, etc.
avaline from an unlimited vocabulary
↳ zones: title, body, etc.

document: free text taken from a dictionary

■ We might assign a weight to each zone so that presence of a query term in each zone of the document contributes as much as the weight
↳ ranked boolean retrieval

→ We can assign a weight parameter to each zone, receive judgement from humans, and calculate our error in terms of the square of the difference between our score & human score. We then choose weights that minimize this error → machine-learned weights.

term frequency: $tf_{t,d}$ = # of occurrences of term "t" in document "d" } a measure of term weight

inverse document frequency: $idf_t = \log(N/df_t)$: # of documents in collection that contain term "t", inversed } used to account for the weight of a term

$tf\text{-}idf_{(t,d)} = tf_{t,d} \times idf_t$

overlap score for document d & query q = $\sum_{t \in q} tf\text{-}idf_{t,f}$

■ Vector Space Model: each document is a vector of term-weights components.

↳ similarity of d_1 & d_2 = $\frac{\vec{V}_{d_1} \cdot \vec{V}_{d_2}}{|\vec{V}_{d_1}| \cdot |\vec{V}_{d_2}|}$ ⇝ dot product / ⇝ Euclidian lengths

↳ gives us a term-document matrix that we can use for similarity measuring.

→ queries can be seen as very short documents and we can rank by document similarity

↳ VERY EXPENSIVE — cosine score

↳ { construct by iterating terms: term-at-a-time
construct by iterating documents: document-at-a-time

Alternate weighting

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

↳ sublinear scaling

② $ntf_{t,d} = a + (1-a)\frac{tf_{t,d}}{tf_{max}(d)}$ ← smoothing factor

↳ max tf normalization

• We can plot document relevance for length buckets against the median length of that bucket and find l_p point at which cosine score collides with it, and rotate it so that it matches our plot better.

↳ Pivoted Normalize Document Length

↳ makes up for the effect of length in document relevance to a query.

ch7

• Instead of doing cosine scoring on all N documents, we can only do it for documents that appear in the postings of query terms, put them in a heap, and get the top K.

• Inexact top-K retrival → ① Select A documents that $K < |A| \ll N$
② Return top-K in A

I index elimination: only consider high "idf" terms, and consider only docs that have all or many query terms → might violate $K<|A|$

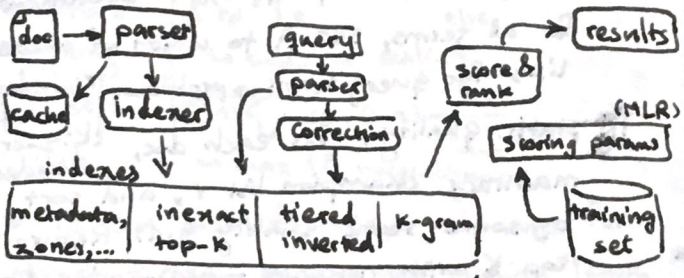
II champion lists: keep top-"r" > k documents for all terms, set A to union of these lists for query terms → problem: { r: indexer decides, k: query decides }

III static quality: for each doc, maintain champion list r, and sort by some static quality g(d). Return top K with net score = g(d) + cosine(d,t)

IV impact ordering: order postings by tf, consider query terms by idf.

V cluster pruning: select $\sqrt{N}$ documents as leaders, connect each follower document to b_1 leaders. At query time consider b_2 clusters close to the query.

* We might create multiple tiers of docs with decreasing tf as champion lists. If tier1 doesn't provide $|A| > k$, we descend to 2, and so on
* Let w be the smallest window in document that contains all query terms → we can try to minimize w to get proximity-weighting
* Query parsers may issue multiple actual queries against the index based on what the user sends by continually relaxing requirements until a set number of results are found ← evidence accumulation.



* we can expand wildcards to associated terms, and then use the vector model.
* if combined carefully, phrase retrieval & vector space retrieval can find phrases while maintaining some notion of a rank.
  ↳ inherently lossy for positional metadata.

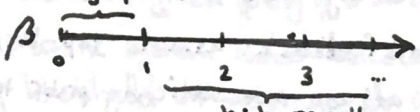
ch8

Having a collection of documents, a number of expressions of information need, and a list relevance judgement for each (need, doc) pair, we can evaluate the search engine.
→ there a number of small to moderately large test suites publicly available to researchers.

* effectiveness (see ch1) / precision = Pr{relevant | retrieved}
  \ recall = Pr{retrieved | relevant}

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

$$F_{\beta=1} = \frac{2PR}{P+R}$$ → used to show the trade off between precision and recall



$$\text{Kappa statistic} = \frac{P(A) - P(E)}{1 - P(E)}$$

$P(A)$: probability of actual agreements
$P(E)$: probability of agreement by chance
↳ normalize judge agreement

* other measures? { responsiveness, indexing speed, user happiness / utility

* including result snippets is important
  ↳ { summarization, highlighting

ch9

*one issue with IR systems is synonymy

Rocchio: $\vec{q}_{i+1} = \alpha \vec{q}_i + \frac{\beta}{|D_r|} \sum_{D_r} \vec{d}_j - \frac{\gamma}{|D_{nr}|} \sum_{D_{nr}} \vec{d}_j$

($\vec{q}_{i+1}$: new query vector; $\vec{q}_i$: previous query; D_r: relevant docs; D_{nr}: nonrelevant docs)

↳ iteratively moves the query vector toward the centeroid of relevant documents.

↳ doesn't work if relevant documents don't cluster nicely.

⇒ relevance feedback works best when we are after improving recall.

↳ not adopted by web users.

- Pseudo Relevance Feedback: assume top-k are relevant and adjust automatically prior to user interaction. → danger of query drift
- Implicit feedback: see which results users click
- Give vocabulary / stemming report to users
- Query expansion by suggesting relevant queries
- Automatically generate a thesaurus
  ↳ often doesn't work well
  - ① term co-occurrence $C = AA^T$, term-doc matrix
  - ② loos grammatical analyses

ch10

- We might want to perform unstructured queries over (semi-)structured data (e.g. XML)
- XML retrieval has many challenges such as what exactly should be returned
- We can use vector space model to map similar documents.
- queries might entail element relationships

ch11

▶ $Pr\{\mathbb{I}=x\} \triangleq P(A) \Rightarrow O(A) = \frac{P(A)}{P(\bar{A})} = \frac{P(A)}{1-P(A)}$ (odds of A)

▶ $P(R=1|d,q)$: the likelihood of "d" being relevant, given query "q" & document "d".

■ If an IR system ranks selected documents by likelihood of relevance to information need according to all available data, the system is at its most effective ⇐ Probability Ranking Principle (PRP)

- 1/0 loss: we lose a point if we return a nonrelevant document or don't return a relevant document
- d is relevant iff $P(R=1|d,q) > P(R=0|d,q)$ (optimal decision rule)
  ↳ works if we know all probabilities.

- We can instead assume a loss/utility model that assigns different costs to false positives & false negatives.

BINARY INDEPENDENCE MODEL : assume terms and documents are independent, and each document is a binary (0/1) vector $\vec{x}$ of terms.

We estimate odds of $\vec{x}$ being relevant to $\vec{q}$ as: $O(R|\vec{x},\vec{q}) = O(R|\vec{q}) \prod \frac{P(x_t|R=1,q)}{P(x_t|R=0,q)}$

$\begin{cases} p_t = \text{term } t \text{ occurring in a doc relevant to } \vec{q} \\ u_t = \text{term } t \text{ occurring in a doc not relevant to } \vec{q} \end{cases}$

$\Rightarrow O(R|\vec{x},\vec{q}) = O(R|\vec{q}) \prod \frac{p_t}{u_t} \prod \frac{1-p_t}{1-u_t}$

▶ Retrieval Status Value (RSV) used for ranking $= \sum_{x_t=q_t=1} \log \underbrace{\frac{p_t(1-u_t)}{u_t(1-p_t)}}_{c_t} = \sum c_t$

▶ In practice: $\log\left(\frac{1-u_t}{u_t}\right) = \log \frac{N-df_t}{df_t} \approx \log \frac{N}{df_t}$

with feedback:

$p_t^{(k+1)} = \frac{|VR_t| + k p_t^{(k)}}{|VR| + k}$ ⇨ can automate by setting $|VR| = |V|$

VR ← members containing t ← relevant results

$|V|$ ← all results

- BM25 is another probabilistic term weighting method that is less presumptive than BIM.
  ↳ more accurate and more popular.
- Bayesian networks can be used to form sophisticated graphical representations of the index & query ← current models make too many assumptions.

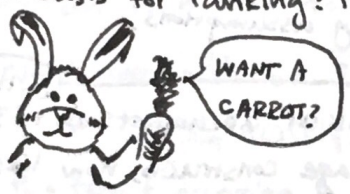
---

ch12

- one way to look at documents is to see based on language constructs, how likely is it for a document to generate a query.
- a probabilistic language model gives us a function which yields the probility of the next term being "t".
- for string "s" and models M_1 & M_2 if $P(s|M_1) > P(s|M_2)$, M_1 is more likely to have generated s.

- chain model: $P(t_1t_2t_3) = P(t_1)P(t_2|t_1)P(t_3|\underbrace{t_1,t_2}_{P_{t,q}})$
- unigram: $P(t_1t_2t_3) = P(t_1)P(t_2)P(t_3)$
- bigram: $P(t_1t_2t_3) = P(t_1)P(t_2|t_1)P(t_3|t_2)$
- probabilistic context-free grammars
- use language analysis

* unigram is usually good enough.
↳ ignores context, proximity, phrases, etc.
↳ bag of words: no ordering
• We generate language models by observing documents, and use generation probability as the basis for ranking: $P(d|q) = \frac{P(q|d)}{P(q)} \cdot P(d)$



this is the same for all documents. usually ignored.

▶ Following these simplifications, we can write $P(d|q) \approx P(q|d) = P(q|M_d)$
↳ using "multinomial unigram language model" we get: $P(q|M_d) = K_q \cdot \prod_{t \in V} P(t|M_d)^{tf_{t,d}}$
↳ multi-nomial coefficient for query (ignored): $K_q = L_d! / (tf_{t_1,d}! \dots tf_{t_n,d}!)$
↳ term freq.

• Since we are using unigram:
$\hat{P}(q|M_d) = \prod_{t \in q} \frac{tf_{t,d}}{L_d}$ ← length of d
▷ the "^" shows estimation.
▷ { - gives too much credit to terms that happened once
- makes it strictly conjunctive
▷ solution: smoothing!

Smoothing

① Adding offset α to avoid zeros & distribute the numbers better
② Linear interpolation: $(0<\lambda<1)$
$\hat{P}(t|d) = \lambda \hat{P}(t|M_d) + (1-\lambda)\hat{P}(t|M_c)$
language model for the whole collection
③ Bayesian Smoothing:
$\hat{P}(t|d) = \frac{tf_{d,t} + \alpha \hat{P}(t|M_c)}{L_d + \alpha}$

• We will mostly use ②
→ it has been shown that the weighting that results from language models out does regular tf-idf (or other related methods).
• we assume documents & queries to be objects of the same type ← same as XML retrieval
• We can construct M_q and try to generate documents from it → weaker estimate
↳ better feedback incorporation
• We can construct M_d & M_q and find out how different they are.
• We can account for "synonymy" by translation models.

ch13

- Standing queries are automatically posed to our indexes periodically.
- we call the process of two-class classification using standing queries "routing" or "filtering."

a.k.a. rules ≈ boolean expressions.

① manual : hard to maintain
② rule-based: inflexible
③ statistical: use machine-learning ↑ start with manually "labeled" set

CLASSIFICATION

For description d in of document space $\mathbb{X}$ about a document, a set of classes $\mathbb{C}$, and training set $\mathbb{D} \subseteq \mathbb{X} \times \mathbb{C}$, we want classification function $\gamma : \mathbb{X} \to \mathbb{C}$

↳ the one-of version, also have any-of where d might be mapped to multiple classes.

■ We denote the supervised learning method resulting in γ as Γ (since we use $\mathbb{D}$)

$$P(c|d) \propto P(c) \cdot \prod P(t_k|c)$$

- using maximum likelihood estimate:

$$\hat{P}(c) = \frac{N_c}{N}, \quad \hat{P}(t_k|c) = \frac{T_{ct}}{\sum T_{ct'}}$$

T_{ct}: t in "c" from training set; $\sum T_{ct'}$: all terms in class "c" from $\mathbb{D}$

- Using add-on (or Laplace) smoothing:

$$\hat{P}(t|c) = \frac{T_{ct} + 1}{\sum T_{ct'} + B}$$

B: # terms in vocabulary.

■ The Bernoulli model: $\hat{P}(t|c) = \frac{N_{c,t}}{N_c}$

↳ doesn't account for multiple occurrences

- To be able to feasibly calculate these values, we make a "~~conditional~~ conditional independence" assumption which means $P(\langle a,b \rangle | c) = P(a|c)P(b|c)$

↳ while not realistic, works well in practice.

- We also assume that in the multinomial model $P(X_{k_1}=t|c) = P(X_{k_2}=t|c)$, and we call it "positional independence".
- The multinomial model is called Naive Bayes (NB) because of its assumptions.

↳ even so, performs really well:

Correct estimation implies accurate prediction, but not the other way around.

- If we replace 0/1 in Bernoulli with term count, it becomes the Naive Bayes model.

▶ Feature Selection: select only a subset of terms, and classify based on those.

☐ ① more efficient training
☑ ② good for classifiers that aren't cheap to train
☐ ③ eliminates noise features → higher accuracy

feature selection

↳ a feature that once added, increases errors
↳ Bernoulli is highly sensitive to noise features

I mutual information: $A(t,c)$ will indicate how much does presence of "t" in a document indicate it belongs to "c".

II χ^2 testing which tests if two events are independent ⇒ even if not statistically strong, it works well with feature selection (independent is ~~bad~~ bad!)

III frequency-based feature selection
based on either document or collection frequency of terms → less accurate, requires many features be selected to work.

↳ These methods can be generalized to n-class classification ← we lose accuracy
• all are "greedy feature selection" methods.
▶ The development set must be kept separate from the training set to avoid biasing the design of the classifier & its decisions.

• Contiguity hypothesis: documents in the same class form a contiguous region & regions do not overlap
↳ used for real-time classification.
-vector space classification depends on proper document representation, weighting the terms, and (length) normalization.
• Each document is vector $\langle e_1, e_2, \dots, e_{|V|} \rangle \in \mathbb{R}^{|V|}$
→ we work with length-normalized vectors that are essentially points in the surface of a hyper-sphere. As long as the arc is small, surface & ~~direct~~ spherical distance are the same: $\lim_{\alpha \to 0} \sin\alpha = \alpha$

• Rocchio classification uses centeroids that are the center of mass for the class. Boundaries are hyperplanes that have each of their points equidistant from the two centeroids they are separating.

$\vec{\mu}(c) = \frac{\sum \vec{v}(d)}{|D_c|}$, $D_c = \{d \mid \langle d, c \rangle \in \mathbb{D}\}$

Rocchio centeroid

• We assign documents to the class whose centeroid is the nearest.
↳ Works well if classes are roughly spherical with the same radii.

Rocchio like NB is $\Theta(|D|.|V|)$
↳ linear

■ Voroni Tessellation partitions a set of objects so that it contains points that are closer to the object and no other.

Then partitions space into $|D|$ convex region containing its corresponding document.

↳ We call this k-nearest neighbors (kNN) method → we assign a document to the majority class of its k closest neighbors.
↳ Choosing the right k is done by experience

nonlinear

⏱ kNN is $\Theta(|D| M_{ave} M_a)$ with preprocessing and $\Theta(|D| L_{ave} M_a)$ without it.
↳ Test time is independent of # of classes.

■ if we use a linear combination of text features & compare it to a threshold for classifying, we are using linear classification.

⇒ We can extend these methods to $J>2$ classes classification

• We first need to know if the classes need to be mutually exclusive.

Classifier performance {
① bias: # of wrong classifications is high
② variance: it assign reliably
}

↳ Classifiers always offer a trade-off between bias & variance.

⇒ There is no universally good classifier

---

ch15

• Support vector machine (SVM) is a linear classifier that insists on a large margin ➡ the distance between the classification line and the nearest data point
↳ This insistence limits the angle that can be used for hyperplanes.
→ a point that is very close to the classification line has close to 50% chance of misclassification.

⇒ The SVM seeks to make high certainty classification decisions.

A classifier is normal vector $\vec{w}$ which is perpendicular to the surface of the classifier hyperplane and constant b.

The SVM can be formalized as a min. problem: find $\vec{w}$ & b such that:

- $\frac{1}{2}\vec{w}^T\vec{w}$ is minimized
- $\forall \{(\vec{x}_i, y_i)\}$: $y_i(\vec{w}^T\vec{x}_i + b) \geq 1$

example ↵ ↳ class label.

- Data points that help specify the separator are called support vectors

↳ The problem is now reduced to a standard quadratic programming problem.

For new vector $\vec{x}$ we project it into the hyperplane & use the sign as ±1 classifier. If distance $\leq b$ we can say "don't know".

The model can also be transformed to a probabilistic model.

SOFT MARAIN CLASSIFICATION: use slack variables to find an optimal margin size that throws away few classless documents.

- The fatter the margin, the better the test time will be. (Fewer support vectors)
- SVMs take $\Theta(|D|^3)$ time to train, and if not trained properly, make many mistakes.
- SVMs can be extended to work with multiple classes, and support structures & relations between classes. (structural SVMs)

■ If the data is nonlinear (= cannot be classified by drawing lines across the hyperspace's surface) we can map it to some higher dimension in which it is.

o SVM classifier: $f(\vec{x}) = \text{sign}(\sum_i a_i y_i \vec{x}_i^T \vec{x}_t + b)$

If we write $k(\vec{x}_i, \vec{x}_j) = \vec{x}_i^T \vec{x}_j$ then f becomes:

$f(\vec{x}) = \text{sign}(\sum_i a_i y_i k(\vec{x}_i, \vec{x}) + b)$

where $k(\vec{x}_i, \vec{x})$ is the future expansion of $\vec{x}$ to a higher dimension.

If k(...) (kernel function) could be computed easily, we wouldn't need to actually transform

Kernel Functions

① Gaussian distribution:
$K(\vec{x}, \vec{z}) = e^{-(\vec{x}-\vec{z})^2/2\sigma^2}$

② Polynomial:
$K(\vec{x}, \vec{z}) = (1 + \vec{x}^T\vec{z})^d$

↳ We can come up with our own, but they have to follow certain criteria.

- Classification has many commercial uses

↳ spam filtering, -

■ When choosing a method, the first thing to ask is the amount of training data available.

↳ never forget the use of standing queries.

• When we train with a limited dataset, & learn over a large unclassified dataset, use have SEMI-SUPERVISED LEARNING.

• ACTIVE LEARNING: have the classifier decide when it should ask humans (usually when uncertain).

⇒ superimposing a rule-based model on a linear classifier makes adjustments to individual classifications very easy.

• IF classifiers make independent mistakes, we can combine them to get more accuracy.

• IF we express d-q relevance as a linear func. of proximity window w & cosine similarity, we can apply learning over a training set to find out the optimal values.

• Once we define classes Φ for expressing whether d_j should precede d_i in the results, we can use machine learning to compose ranking functions.

↳ Still not very commercial and mostly theoretical.

ch 16

CLUSTERING as opposed to classification is completely unsupervised. ⇒ distance is the guide.

① Flat vs Hierarchical

② Hard vs Soft
exactly one cluster per doc ↳ multiple clusters per doc

💡 = Cluster hypothesis: documents in the same cluster behave the same for similar information needs in terms of relevance.

→ Search result clustering (meanings in Duckduckgo)

→ Scatter-Gather: iteratively cluster and descend into clusters.

→ Expand recall by adding documents that are in the same cluster as relevant results.

→ Improve ranking and recall speed by only considering documents from clusters that are close to the query.

■ Given $D=\{d_0,\dots,d_{n-1}\}$, clusters K, objective function that judges cluster quality, we want to find $Y: D \to \{1..k\}$ that is minimum for the objective function.
↳ often similarity or distance

- Cardinality of a clustering (K) is an important performance & accuracy factor.

Standard Quality Criteria
① Internal: judge based on co-relevance of each cluster's docs.
② External: use gold standard from human judges.

External Measures:

① purity: assign the cluster to its majority class and evaluate the assignment

② normalized mutual information (NMI):

clusters ← Ω , C → classes

$$NMI(\Omega, C) = \frac{I(\Omega; C)}{[H(\Omega)+H(C)]/2}$$

→ mutual info (see ch13)

entropy (see ch5)

③ Rand Index (RI) ≈ accuracy

	positive	neg
true	TP	TN
false	FP	FN

$$RI = \frac{TP+TN}{TP+TN+FP+FN}$$

④ F-measure (see ch 8):

$$F_\beta = \frac{(\beta^2+1)PR}{\beta^2 P+R} \quad ; \quad P = TP/(TP+FP) \quad R = TP/(TP+FN)$$

→ $\Theta(|D|)$

K-means minimize average of document distance from cluster centeroid:

$$\vec{\mu}(\omega) = \frac{1}{|\omega|}\sum_{\vec{x}\in\omega}\vec{x} \quad \text{for cluster } \omega$$

- Residual Sum of Squares (RSS):

$$RSS_k = \sum_{\vec{x}\in\omega_k} |\vec{x}-\vec{\mu}(\omega_k)|^2$$

$$RSS = \sum_{k=1}^{K} RSS_k \rightsquigarrow \text{this is the objective function which we want to minimize for } K$$

HOW? Select K clusters' centeroids at random (seeds) and assign RSS, and keep reassessing centeroids until a quit condition is met:
- a set number of iterations
- γ doesn't change
- $\vec{\mu}_k(\omega_k)$ doesn't change
- RSS ≤ threshold

RSS is evaluated to an approximation of global maximum.

⇒ finding the best K is difficult.

Using distortion model complexity criteria, by considering RSS as distortion & f(K) (usually f(K) = K) as model complexity we have: $K = \underset{\forall K}{\text{argmin}}\,[RSS_{min}(K) + \lambda K]$

- Larger λ means fewer clusters.

But how do we find λ?

- use previously successful values.

- $\lambda = 2M$ ↝ dimensionality of the vector space
  ↳ but this can easily lead to $2Mk$ term dominating the identity & thus leading to a situation where $k \to !\bullet$.

■ Model-based clustering assumes the data (documents) were generated by a model (by adding random noise to it), and tries to recover the original model, which becomes the cluster.

Let's call model parameters θ ($= \vec{\mu}_1, \ldots, \vec{\mu}_{k\text{-means}}$ in k-means)

$\theta = \arg\max_\theta L(D|\theta) = \arg\max_\theta \sum_i^N \log P(d_i|\theta)$

↳ softly clusters, since we have $P(d|\omega_k;\theta)$

↳ $L(D|\theta)$ is the objective function

→ $\theta(|D|)$

► EM (expectation maximization) tries to iteratively maximize $L(D|\theta)$, by reassigning its random parameters & gauging its effect on the objective function $L(D|\theta)$ to finally reach a satisfactory conclusion.

⇒ The most essential part is finding good seeds.

ch17

issues with Flat clustering { - not enough structure; - predetermined # of clusters; - non-deterministic

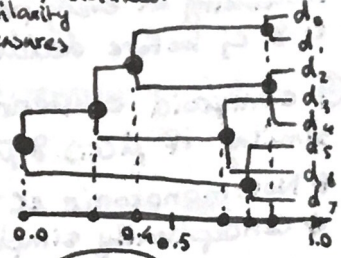
↳ we use hierarchical clustering if we are concerned → usually slower & less efficient

Hierarchies: { - top-down: form clusters, break them apart recursively; - bottom-up: cluster per doc, merge upwards } agglomerative ←

→ a framework with specialized similarity measures

HAC

Hierarchical Agglomerative Clustering



dendrogram of cluster merge points by similarity ←

shows history of mergers ⇐ time

* If we want disjoint clusters, or a pre-specified # of clusters:
  ① cut at similarity s. e.g. 0.4 yields $k=8$
  ② cut when gap is large (natural clustering point)
  ③ apply distortion model complexity
  ④ stop at prespecified k^{th} cluster

Flavors of HAC:

① Single-link clustering: gauge similarity of two clusters by their most similar member
↳ usually skewed :($\Theta(N^2)$

② Complete-link clustering: gauge similarity by their least similar members ⇒ creates clusters with smaller diameters 👍
• pays too much attention to outliers :(

③ (group-)average clustering: measures the similarity of every pair of document in c_i & c_j before deciding to merge.

④ centeroid clustering: considers c_i & c_j similar if $\vec{\mu}(c_i)$ & $\vec{\mu}(c_j)$ are similar.
◆ Not monotonic :(⇒ not optimal
◆ Conceptually simpler 👍

① ② ③ ④ (N² log N)

TOP-DOWN or "divisive" clustering

- Divide the space into K clusters and recursively divide each cluster until we have singleton clusters (=single-doc).
- Use a flat clustering algorithm (eg K-mean)
- If we stop at "m" levels: $\Theta(N)$ 👍
- conceptually harder :(

Labeling the clusters:
① Differential: use feature selection
↳ considers the collection as a whole
② Internal: consider "title", or most frequent term → only cares about the local maxima.

---

ch18

- For matrix $C_{M\times N}$, rank(C) is the # of linearly independent rows (or columns)

$\text{rank}(C) \le \min\{M,N\}$

- C is diagonal if its r×r & all off-diagonal entries are zero
- rank of a diagonal matrix is # of its non-0 diagonal entries.

For $C_{r\times r}$ & $\vec{x}$ that is not all zeros:

$$C\vec{x} = \lambda\vec{x}$$

eigenvalues of C ← λ; → right eigenvector of C

$$\vec{x}^T C = \lambda\vec{x}^T$$

→ left eigenvector of C

# of nonzero eigenvalues of C ≤ rank(C)

$\det(C-\lambda I_r) = 0$ ⇒ gives us eigenvalues of C

• arbitrary vectors can be expressed as linear combination of eigenvalues ⇒ makes multiplication $M\vec{x}$ irrelevant of $\vec{x}$

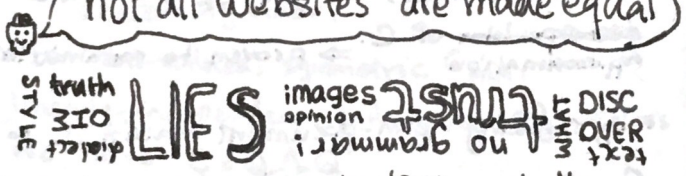
- If we write $\vec{x}$ in terms of λ, and ignore smaller terms, we get $\vec{y}$ that is "close" to $\vec{x}$.
- For C symmetric & real, the eigenvectors are orthogonal.
- For $C_{M \times M}$ real-valued with M linearly-independent eigenvectors, C can be written as $C = U \Lambda U^{-1}$
  - columns of U are eigenvectors of C
  - Λ is diagonal with eigenvalues of C in decreasing order.
- For real-valued, symmetric $C_{M \times M}$, with M linearly-independent, orthogonal eigenvectors we have $C = Q \Lambda Q^T$
  - Q is the concatenation of all eigenvectors of C, length-normalized
  - $Q^{-1} = Q^T$
- For $C_{M \times N}$, make $U_{M \times M}$ from eigenvectors of CC^T, and $V_{N \times N}$ from eigenvectors of $C^T C$. Then $C = U \Sigma V^T$ ~ Singular Value Decomposition (SVD)
  - eigenvalues of CC^T & $C^T C$ are equal
  - $r = \text{rank}(C) \Rightarrow$ For $1 \le i \le r$: $\sigma_i = \sqrt{\lambda_i}$ where $\lambda_i \ge \lambda_{i+1}$.
  - $\Sigma_{ij} = \begin{cases} \sigma_i & i = j \le r \\ 0 & \text{otherwise} \end{cases}$

- For C, if we create C_k where $\text{rank}(C_k) \le k$ we wish to find C_k that minimizes: $\|X\|_F$ (Frobenius norm) $= \sqrt{\sum_{i=1}^{M} \sum_{j=1}^{N} x_{ij}^2}$, $X = C - C_k$
- Finding C_k:
  1. take SVD of C; $C = U \Sigma V^T$
  2. replace $r-k$ smallest values of Σ with 0 to get Σ_k
  3. compute $C_k = U \Sigma_k V^T$
- if $k \ll \text{rank}(C)$ we call C_k a low-rank ~~decomposition~~ approximation of C. $\Rightarrow$ proven to minimize $\|X\|_F$
- By reducing term-document matrix C to C_k for some k, we achieve latent semantic indexing (LSI).
- Computing SVDs is taxing.
- LSI can improve recall and by handling synonymy can even improve precision.
- Its scalability is questionable, but other than that, it can force documents to rank similarity according to collocation
- $\vec{q}_k = \Sigma_k^{-1} U_k^T \vec{q}$ ~ bringing query q into C_k space.

ch19

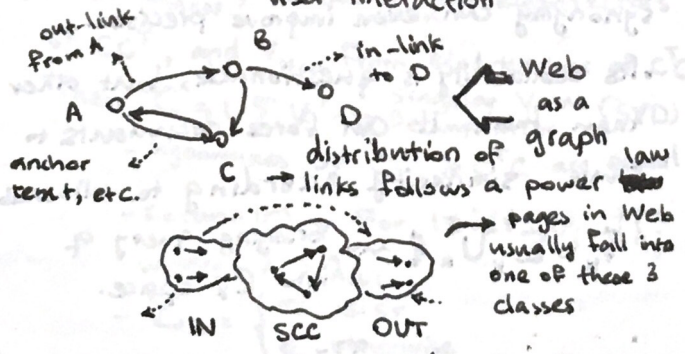
- Allowing browsers to be lax with markup encouraged amateur content creators.
- Early attempts at making Web content discoverable
  1. Index-based search engines → Altavista
  2. taxonomies → Yahoo!
- Early search engines overcame scalability issues only to face result quality issues

not all websites are made equal



- getting to contents itself is a challenge, let alone judging which one is more trustworthy

Web pages → static: unlikely to change from visit to visit
→ dynamic: usually the result of user interaction



SPAM → adversarial information retrieval

1. Creating content that abuses term-based relevance ranking.
   ↳ even hiding it by styling
2. Cloaking: serve a different content to search engines vs. humans.
3. Doorway page: redirect humans to a different page
4. Link spam: abuse link analysis techniques to rank higher.

* Some search engines allow for various forms of paid inclusion

ads
- cost per mil: money for displaying an ad 1000 times → brand promotion
- cost per click → transaction-oriented

↳ Goto/Overture took bids on queries and returned results ranked by bids.

▶ Google started doing the same thing, combining actual & paid-for results in some form in the user interface.

UX
1. Focus on precision of first few results
2. lightweight & free-form user interface

Google principles for attracting traffic & generating revenue

- Web users are regular people that don't know about indexes & operators.

Web Searches

① Informational: user is seeking information on a topic
② Navigational: user is looking for a specific web destination
③ Transactional: user is looking for service/goods providers.

Estimating which search engine has indexed more of the is difficult.
We can submit random queries to E_1 & E_2 to see which of them returns more comprehensive results.

Another challenge is detecting near-duplicates.
- d_i & d_j are similar if their k-shingles are.
  ↳ use Jaccard (see ch3) → expensive
  → compute a hash of each document, then sample the index (sketch) and compare with their hashes.

a b c d e
3-shingles

---

ch20

Web crawler (spider) gathers pages from the internet alongside its link structure.

CRAWLER
- must → robust: avoid traps
- must → polite
- should → distributed
- should → scalable
- should → performant: be efficient
- should → continuous: refresh pages
- biased by quality
- extensible

A crawler must consult the 'robots.txt' file for the sake of politeness.
↳ sometimes cannot be cached since it may change, but most of the times it can.
- Also, the crawler must get rid of duplicates via hashing or shingling.
- URs are read from a URL frontier which contains a prioritized list of pages to fetch
  - URLs also need to be normalized to some canonical form.
  - Distributing the set of URLs already explored as well near-duplication detection modules is very challenging.
  - Most crawlers need to have their own DNS component to implement a non-blocking shared DNS lookup scheme.
  - The URL frontier is made of back queues that enforce politeness & front queues that enforce priority, with an element of randomness
- index distribution / by term: non trivial / by document: most used
- We need connectivity servers that cache connectivity (edge-level) queries between pages with memory-friendly compression that does not add query overhead.

ch21

- anchor texts can be used as descriptors of the page they are pointing to.
- we can also consider a window of text surrounding the anchor text as an extension..
- If a surfer was to randomly follow hyperlinks from a starting point & whenever stuck, jump to any other random page, they are likely to visit some pages more frequently ⇒ PageRank

A matrix $\begin{cases} \forall i,j\ P_{ij} \in [0,1] \\ \forall i \sum_j P_{ij} = 1 \end{cases}$ is called a "stochastic matrix. ⇒ always have a principal left eigenvector

Markov chains are stochastic processes that happen through discrete time & involve making choices at each juncture.
↳ next move only depends on the current state

Forming the Markov Chain Matrix
① Create $M_{N \times N}$
② $M[i,j] = \begin{cases} 1 & \text{if } i \text{ links to } j \\ 0 & \text{otherwise} \end{cases}$
③ if row is all zero set all elements to $1/N$
④ divide 1s by the # 1s in row
⑤ multiply M by $(1-\alpha)$
⑥ add α/N to every element

α: teleport probability

M is ergodic if $\exists T_0 > 0$: if at t_0 we are at "i", at $t \geq T_0$ $P(i,j)$ $\forall j > 0$

For any ergodic matrix for a Markov chain a unique steady state vector $\vec{\pi}$ (= principal left eigenvector of M) exists. If $\eta(i,t)$ is # of visits to i after t steps:

$\lim_{t \to \infty} \frac{\eta(i,t)}{t} = \pi(i)$ → steady-state probability for state i

$\vec{\pi} = \lim_{t \to \infty} \vec{x}_0 P^t$ ⇒ iterative power is one way of calculating PageRank as at some step t_i P^{t_i} & P^{t_i+1} are bound to converge.

By overlaying P with a user-based teleportation to specific topics, we can involve user preferences ⇒ hard to implement.

Pages for informational queries
① authorities in the topic of query
② crafted pages that are hub of links to authority pages

We give each page two scores:

* hub score: $h(v) \leftarrow \sum_{v \to y} a(y)$
* authority score: $a(v) \leftarrow \sum_{y \to v} h(y)$

and then unroll these to make it non-recursive.

Consider A to be adjacency matrix of our subset of the web.

Then we have: $\begin{cases} \vec{h} \leftarrow A\vec{a} \\ \vec{a} \leftarrow A^T\vec{h} \end{cases}$ and by substitution we get this: $\begin{cases} \vec{h} \leftarrow AA^T\vec{h} \\ \vec{a} \leftarrow A^TA\vec{h} \end{cases}$

which is extremely similar to eigenvalues:

$$\begin{cases} \vec{h} = (1/\lambda_h)\, AA^T\vec{h} \\ \vec{a} = (1/\lambda_a)\, A^TA\vec{a} \end{cases}$$

- We now only need to compute λ_h & λ_a which can be done using any of the available methods.

Hyperlink-induced Topic Search

or HITS is this process of applying the above method to queries & getting $\vec{a}$ & $\vec{h}$.

① Select a "root" set from the index for "q"

② Include pages with in- & out-links to root set and call it base set.

③ Compute HITS for base set.

↳ even crosses languages since after selecting root set we discard the query.