

Building strings are arrays of proper
subtrees at a given height.

EXAMPLES:
An array
An matrix
An tree
An list
An set

} We can see that
building strings are
than comparing two
adjacent nodes pairwise
can yield the longest
repeated substring of S.

► Generating random test & searching for
patterns among the trees of subtrees among
the building strings.

► Generating random test & searching for
patterns among the trees of subtrees among
the building strings.

► Generating random test & searching for
patterns among the trees of subtrees among
the building strings.

► Generating random test & searching for
patterns among the trees of subtrees among
the building strings.

► Generating random test & searching for
patterns among the trees of subtrees among
the building strings.

► Generating random test & searching for
patterns among the trees of subtrees among
the building strings.

► Generating random test & searching for
patterns among the trees of subtrees among
the building strings.

Introduction to Information Retrieval

2
0
0
9

A term-document matrix helps in information retrieval → especially

Christopher D. Manning

Prabhakar Raghavan

↳ We can measure quality of search results

precision/recall → precision how many of what is relevant → recall how much of what is relevant is recalled

→ the document matrix is extremely sparse

→ we can build document vectors → an inverted index which has the same problem of being sparse → by normalizing it to unit length

ch 1 - ch 21

44 pages

2016, 11, 17 - 2016, 11, 20

~~2~~ | of behavior
~~3~~ | ~~reaction~~
~~4~~ | ~~behavior~~

Characteristics
of adaptation

Ex No - 10

2000, 2001

26, 11, 2002 - 27, 11, 2002

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

broaden than word → document
 A term-document incidence matrix helps in "boolean" retrieval → operators $\in \{\text{AND}, \text{OR}, \text{NOT}\}$

(→ 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?

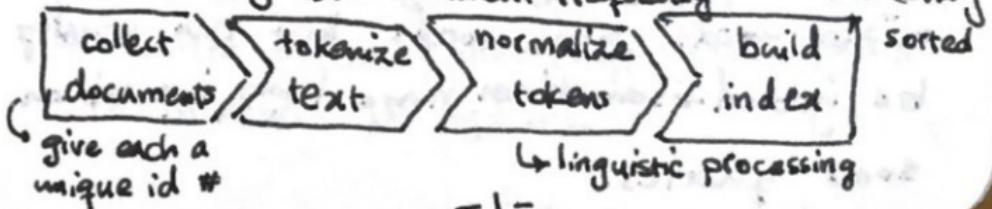
⇒ the incidence matrix is extremely sparse

term₀ → doc₀ → doc₁ → ...
 term₁ → doc₀ → doc₁ → ...
 ...
 term_n → doc₀ → doc₁ → ...

posting list

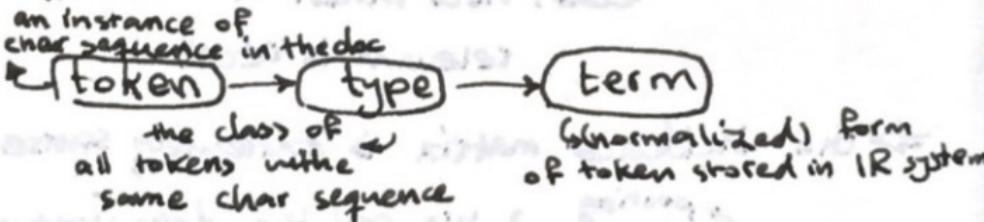
size of posting list = document frequency

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



- To process boolean queries, we evaluate each conjunctive term in increasing document frequency, and calculate the intersection \rightarrow linear merge $O(N)$.
- The extended boolean model also adds a "proximity" operator (# of intervening words, or some structural restriction, e.g. same sentence).
 - \rightarrow AND increases precision, decreases recall
 - \rightarrow 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.



- We process the queries with the same normalizers as we used with the documents.
- ~~normalization~~
 - sometimes instead of trying to discover words, we might just use k-grams.
 - "stop words" are words that can usually be ignored \rightarrow 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)

	<u>stemming</u>	<u>lemmatization</u>
saw(v)	s	see
saw(n)	s	saw

→ neither offers holistically great gains to modern IR systems.

- We can use skip pointers to speed up merging \rightarrow great if indexes are static
 - \hookrightarrow \sqrt{P} pointers for posting lists of size $l \cdot P$

Positional Postings & Phrase Queries

larger indexes \leftarrow
more complex \leftarrow
intersections

$\sim 26\%$ more space, 75% speed-up

- ① use biwords $\rightarrow xyz \Rightarrow \{(n, j) | y, z\}$
 - \hookrightarrow avoid false positive by k-words
 - \hookrightarrow extended biwords by part-of-speech tagging: head over heads $\Rightarrow (\text{head}, \text{heads})$
 - Variable K: we have a phrase index
 - \hookrightarrow gets exponentially large
- ② positional indexes: store each occurrence of the term in each document \rightarrow this enables us to do proximity queries.
- ③ store positions but for common phrases and words with high document frequency store biwords
- ④ keep a partial next-word index

ch 3

looking up the terms requires a dictionary data structure

- | | |
|----------------|-------------------------|
| ① hashing: see | CLRS(14) |
| | ② binary tree: CLRS(18) |
| | ③ B-trees: see CLRS(27) |

- hashing isn't great in the face of an ever-growing vocabulary, doesn't let iterations
- balancing binary trees is expensive
 - \hookrightarrow 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\$ → well → postings
and rotate ellswe → well
query. l\$wel
↳ prohibitively large

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

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

abc✓

Spell Correction

isolated → edit distance
(Levenshtein)

↳ use Jaccard coefficient $\frac{|A \cap B|}{|A \cup B|} < k$ to find candidates from corpus first

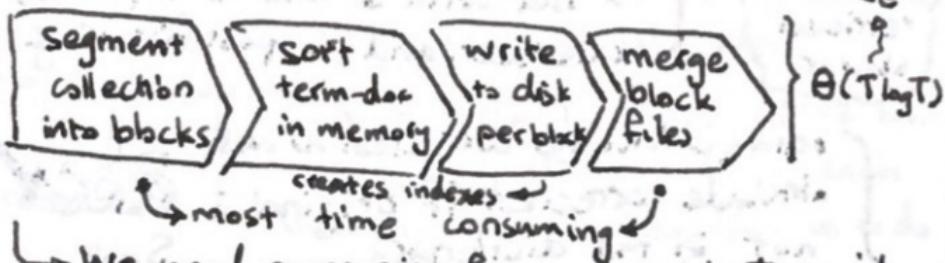
context-sensitive → correct all words and query each result to find max phonetic correction : use soundex

herman → "H655"

↳ uses equivalence classes for similar-sounding consonants

- The design of indexers 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 \rightarrow we buffer
- CPU is free while reading \Rightarrow reading compressed data is faster

BLOCKED SORT-BASED INDEXING (BSBI) collection size



\hookrightarrow We need a mapping from term to term id

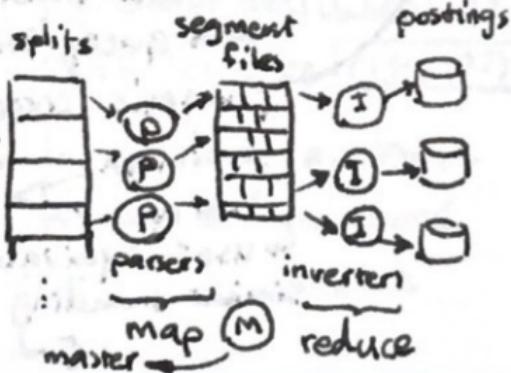
SINGLE-PASS IN-MEMORY INDEXING (SPIMI)

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

\hookrightarrow memory stuff & disk files can be zipped.

\hookrightarrow no sorting $\Rightarrow \Theta(T)$ term partitions

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. $\Theta(T^2/n)$ $\xrightarrow{\text{total postings}}$ $\xrightarrow{\text{memory index size}}$

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

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

↳ Indices 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.

compressing the **INDEX**

- ① less disk space (duh!) ch5
② better caching \leftarrow more things fit in RAM
③ faster transfer \leftarrow decompression overhead is worth it!

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

► compression { lossy \rightarrow when lost information is unlikely to be searched for
lossless

HEAP'S LAW: $M = kT^b$, $30 \leq k \leq 100$, $b \approx 0.5$
number of terms \leftrightarrow number of tokens

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



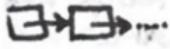
① treat it as one large string and use pointers to end of words. Locate by binary searches $\approx 60-30\%$ decrease

COMPRESSING THE DICTIONARY ② cut the string into k-term blocks
 $k \searrow$ decrease: better lookup, larger dictionary

\nwarrow 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
 \nwarrow 7-bit payload
 \nwarrow very effective, easy to implement

② Y code: 1110 101

unary representation \nwarrow offset with leading
of offset length '1' removed

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



Fields: author, date, etc.

values 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 : # of documents in collection that contain term "t"; inversely } used to account for the weight of a term

 $= \log(N/df_t)$

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

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

document $-q-$ query

- Vector Space Model: each document is a vector of term-weight components.
 - ↳ similarity of d_1 & d_2 = $\frac{\vec{v}_{d_1} \cdot \vec{v}_{d_2}}{|\vec{v}_{d_1}| \cdot |\vec{v}_{d_2}|}$ \rightsquigarrow dot product \rightsquigarrow Euclidean lengths
 - ↳ gives us a term-document matrix that we can use for similarity measuring.
 - \rightarrow 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
- Alternative Weighting Smoothing factor
- $$\text{① } wF_{td} = \begin{cases} 1 + \log tF_{td} & tF_{td} > 0 \\ 0 & \text{else} \end{cases}$$
- ↳ sublinear scaling
- $$\text{③ } ntf_{td} = \alpha + (1-\alpha) \frac{tF_{td}}{tF_{\max}(d)}$$
- ↳ 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 Normalized Document Length
- ↳ makes up for the effect of length in document relevance to a query.

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 retrieval \rightarrow
 - 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 \rightarrow might violate $K \leq N$

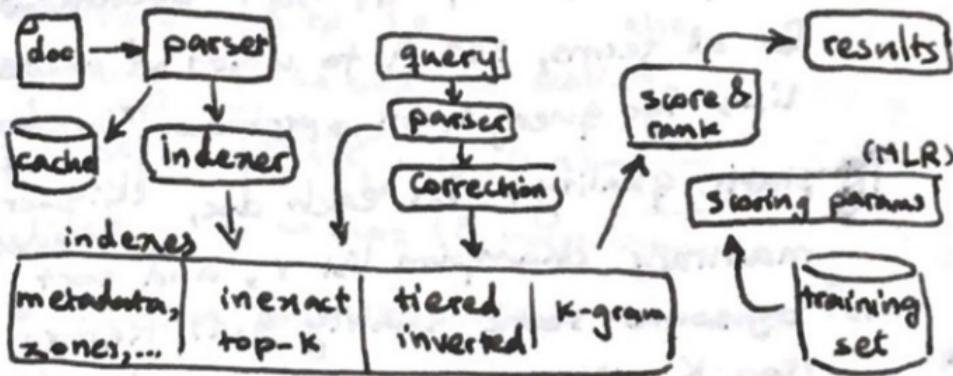
II champion lists: keep $\text{top-}^{\text{r}} > k$ documents for all terms, set A to union of these lists for query terms \rightarrow problem: r: indexer decides

III static quality: for each doc, c_k : query decides maintain champion list r, and sort by some static quality $g(d)$. Return top K with net score = $g(d) + \text{cosine}(d, t)$

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 b₁ leaders. At query time consider b₂ clusters close to the query.

- * We might create multiple tiers of docs with decreasing tf as champion lists. If tier 1 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 \rightarrow 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 \leftarrow 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.

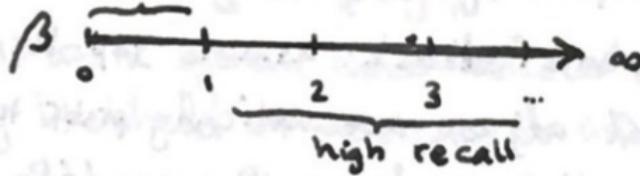
Having a collection of documents, a number of expressions of information need, and a list relevance judgement for each (need, doc) pair, we can evaluate the search engine.
 → there are a number of small to moderately large test suites publicly available to researchers.

effectiveness / precision = $\frac{\text{Pr}\{\text{relevant} \mid \text{retrieved}\}}{\text{Pr}\{\text{retrieved}\}}$
 (see ch1) recall = $\frac{\text{Pr}\{\text{retrieved} \mid \text{relevant}\}}{\text{Pr}\{\text{retrieved}\}}$

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

$$F_{\alpha=1} = \frac{2 PR}{P+R}$$

↳ used to show the trade off between precision and recall



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

↳ normalize judge agreement

↳ probability of actual agreements
 ↳ probability of agreement by chance

* Other measures? { responsiveness
 indexing speed
 user happiness/utility

* Including result snippets is important
 ↳ { summarization
 highlighting

* one issue with IR systems is synonymy

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

new ↗ previous ↗ relevant ↗ nonrelevant
 query ↗ query ↗ docs ↗ docs

↳ iteratively moves the query vector toward the centroid 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 user
- Query expansion by suggesting relevant queries
- Automatically generate a thesaurus
 - ↳ often doesn't work well
 - ① term co-occurrence matrix
 - ② uses grammatical analyses

- 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

$$\Pr\{I=X\} \triangleq P(A) \Rightarrow O(A) = \frac{P(A)}{\text{odds or } A} = \frac{P(A)}{1-P(A)}$$

► $P(R=1 | d, q)$: 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 \leftarrow Probability Ranking Principle (PRP)

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

$$O(R|\vec{x}, \vec{q}) = O(R|\vec{q}) \prod \frac{P(x_t | R=1, \vec{q})}{P(x_t | R=0, \vec{q})}$$

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

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

► Retrieval Status Value (RSV) used for ranking

$$\text{ranking} = \sum_{t=1}^n \log \underbrace{\frac{P_t(1-U_t)}{U_t(1-P_t)}}_{C_t} = \sum C_t$$

► 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:

$$\overbrace{\text{VR}}^{\substack{(k+1) \\ \text{members} \\ \text{containing} \\ \text{relevant} \\ \text{results}}} = \frac{|VR_t| + kP_t^{(k)}}{|VR| + k}$$

can automate
by setting
 $|VR| = |V|$
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 probability 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.

$$\left\{ \begin{array}{l} \text{- chain model : } P(t_1 t_2 t_3) = P(t_1) P(t_2 | t_1) P(t_3 | t_2) \\ \text{- unigram : } P(t_1 t_2 t_3) = P(t_1) P(t_2) P(t_3) \\ \text{- bigram : } P(t_1 t_2 t_3) = P(t_1) P(t_2 | t_1) P(t_3 | t_2) \\ \text{- probabilistic context-free grammars} \\ \text{- use language analysis} \end{array} \right.$$

- * unigram is usually good enough.
 - ↳ ignores context, proximity, phrases, etc.
 - ↳ bag of words: no ordering

- We generate language models by observing documents, and we generate probability \approx the basis for ranking: $P(d|q) = \frac{P(q|d)}{P(q)} \cdot P(d)$



this is \approx
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):

$$k_q = L_d! / (tf_{1,d}! \cdots tf_{n,d}!)$$

- Since we are using unigram:

$$\hat{P}(q|M_d) = \prod_{t \in q} \frac{tf_{t,d}}{L_d}$$

↳ the " $\hat{}$ " shows estimation.
 L_d is length of d

- { -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 outdoes 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.

- 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 \propto boolean expressions.



① manual : hard to maintain

② rule-based: inflexible

③ statistical: use machine-learning

CLASSIFICATION ↗ start with manually "labeled" set

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

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

We denote the supervised learning method resulting in γ as Γ .

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

using maximum likelihood estimate:

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

$\sim t$ in "c" from training ser

all terms in class 'c' from ID

Solution: Smoothing

- Using add-on (or Laplace) smoothing:

$$\hat{P}(t|c) = \frac{T_{ct} + 1}{\sum T_{ct} + \#B \rightsquigarrow \text{# terms in vocabulary.}}$$

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

(\rightarrow doesn't account for multiple occurrences)

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

(\rightarrow 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.

\hookrightarrow 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 by its decisions.

- Contiguity hypothesis: documents in the same class form a contiguous region & regions do not overlap
 ↳ used for real-value classification.
 - vector space classification depends on proper document representation, weighting the terms, and (length) normalization.

- Each document is vector $\langle e_1, e_2, \dots, e_m \rangle \in \mathbb{R}^m$
 → 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 & ~~spherical~~ distance are the same: $\lim_{\alpha \rightarrow 0} \sin \alpha = \alpha$



- Rocchio classification uses centroids that are the center of mass for the class.

Boundaries are hyperplanes that have each of their points equidistance from the two centroids they are separating.

$$\vec{M}(c) = \frac{\sum \vec{v}(d)}{|D_c|}, D_c = \{d | (d, c) \in D\}$$

Rocchio centroid

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

Rocchio like NB is $\Theta(|D| \cdot |V|)$

↳ linear ↳

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

Then partitions space into $|D|$ convex regions 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

↳ kNN is $\Theta(|D| M_{avg} M_n)$ with preprocessing and $\Theta(|D| L_{avg} M_n)$ 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

nonlinear

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 assigns reliably
↳ classifiers always offer a trade-off between bias & variance.

⇒ There is no universally good classifier

-
- Support vector machine (SVM) is a ch15 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 90° 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 ± classifier.

If distance $\leq b$ we can say "don't know".

The model can also be transformed to a probabilistic model.

☞ SOFT MARGIN CLASSIFICATION: we 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 $\Theta(1DI^3)$ 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 : $f(\vec{x}) = \text{sign}(\sum_i \alpha_i y_i \vec{x}_i^T \vec{x} + b)$

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

$$f(\vec{x}) = \text{sign}\left(\sum_i \alpha_i y_i K(\vec{x}_i, \vec{x}) + b\right)$$

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

If $K(\dots)$ (kernel function) could be computed easily, we wouldn't need to actually transform

① Gaussian distribution



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

Kernel ② Polynomial:

$$\text{Functions } K(\vec{x}, \vec{z}) = (1 + \vec{x}^T \vec{z})^d$$

↳ 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 undclassified dataset, we 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 classifier's make independent mistakes, we can combine them to get more accuracy.
- If we express d_j -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 \mathbb{I}_1 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.

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

① Flat vs Hierarchical

② Hard vs Soft

exactly one cluster per doc \hookrightarrow multiple clusters per doc

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

\rightarrow Search result clustering (meaning in Duckduckgo)

\rightarrow Scatter-Gather: iteratively cluster and descend into clusters.

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

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

 Given $D = \{d_1, \dots, d_m\}$, clusters K , objective function that judges cluster quality, we want to find $Y: D \rightarrow \{1 \dots k\}$ that $\min_{\text{similarity}}$ For the objective function,

\hookrightarrow 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):

$$NMI(\Omega, C) = \frac{I(\Omega; C)}{\text{entropy (see ch 5)} + [H(\Omega) + H(C)]/2} \rightarrow \begin{matrix} \text{clusters} \leftarrow \\ \text{classes} \end{matrix} \rightarrow \begin{matrix} \text{mutual info} \\ (\text{see ch 13}) \end{matrix}$$

③ Rand Index (RI) \approx accuracy

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

	positive		negative
true	TP	TN	
false	FP	FN	

④ F-measure (see ch 8):

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

$$R = TP / (TP + FN)$$

$\rightarrow \theta(101)$

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

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

■ Residual Sum of Squares (RSS):

$$RSS_k = \sum_{\tilde{x} \in w_k} |\tilde{x} - \bar{\mu}(w_k)|^2$$

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

HOW? Select K clusters' centroids at random (seeds) and assign RSS, and keep reassessing centroids until a quit condition is met:

- a set number of iterations
RS is evaluated → - $\bar{\mu}(w_k)$ doesn't change
to an approximation of global maximum. - $RSS \leq \text{threshold}$

⇒ 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: $K = \underset{K}{\operatorname{argmin}} [RSS_{\min}(K) + \lambda K]$

• Larger λ means fewer clusters.

But how do we find λ ?

◦ use previously successful values.

- $\lambda = 2M \rightsquigarrow$ dimensionality of the vector space
 - ↳ but this can easily lead to $2MK$ term dominating the identity & thus leading to a situation where $K \rightarrow 0$.
- 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 ($= t\hat{\mu}_1, \dots, t\hat{\mu}_k$)

$$\theta = \underset{\theta}{\operatorname{argmax}} L(D|\theta) = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^N \log P(d_i|\theta)$$

↳ softly clusters, since we have $P(d_i|w_k; \theta)$

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

- 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.

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

a framework with specialized similarity measures

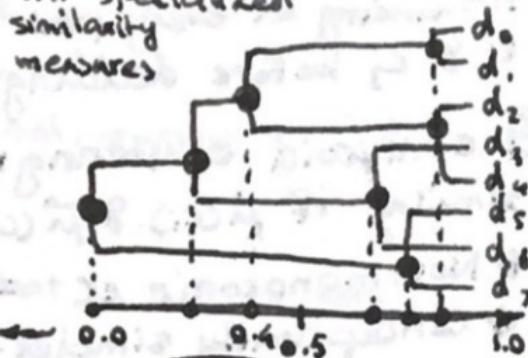
HAC

Hierarchical Agglomerative Clustering

dendrogram of

cluster merge

points by similarity



Shows history of merges

time ←

* IF we want disjoint clusters, or a pre-specified # of clusters

- ① cut at similarity s_* . e.g. 0.4 yields $k=9$
- 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 :c $\Theta(N^2)$
- ② Complete-link clustering: gauge similarity by their least similar members \Rightarrow creates clusters with smaller diameters \clubsuit
 - pays too much attention to outliers :c
- ③ (group-)average clustering: measures the similarity of every pair of document in c_i & c_j before deciding to merge.
- ④ centroid clustering: considers c_i & c_j similar if $\bar{\mu}(c_i)$ & $\bar{\mu}(c_j)$ are similar.
 - ◆ Not monotonic :c \Rightarrow not optimal
 - ◆ Conceptually simpler \clubsuit

⚠ 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: $\Theta(N)$ \clubsuit
- conceptually harder :c

Labeling the Clusters {

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

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

- C is diagonal if its $r \times r$ & 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 λ ← right eigenvector of C
of C ← left eigenvector of C

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

of nonzero eigenvalues of $C \leq \text{rank}(C)$

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

- arbitrary vectors can be expressed as linear combination of eigenvalues \Rightarrow 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 $C_{M \times n}$ real-valued with n 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^TC .
 Then, $C = U \Sigma V^T$ \approx Singular Value Decomposition
 - eigenvalues of CC^T & C^TC are equal
 - $r = \text{rank}(C) \Rightarrow$ For $1 \leq i \leq r$: $\sigma_i = \sqrt{\lambda_i}$
 where $\lambda_i > \lambda_{i+1}$.
 - $\sum_{ii} = \begin{cases} \sigma_i & 1 \leq i \leq r \\ 0 & \text{otherwise} \end{cases}$

- For C , if we create C_k where $\text{rank}(C_k) \leq k$ we wish to find C_k that minimizes:

Frobenius norm

$$\|X\|_F = \sqrt{\sum_{i=1}^M \sum_{j=1}^N x_{ij}^2}, X = C - C_k$$

- 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 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.

$$\tilde{q}_k = \sum_{i=1}^k U_k^T q \rightsquigarrow \text{bringing query } q \text{ into } C_k \text{ space.}$$

- 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



truth
type
3IO
dilect

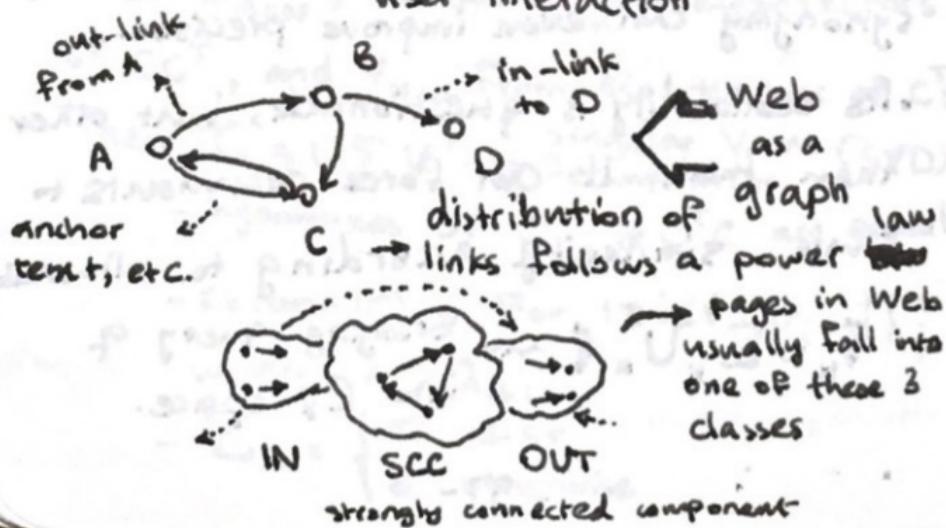
LIES

images
opinion
grammar
on

DISC
OVER
text

- 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

retrieval
information
processing

- 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

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

& generating revenue



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 $\underline{\underline{a \ b \ c}} \ \underline{\underline{d \ e}}$
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.

ch20
Web crawler (spider) gathers pages from the internet alongside its link structure.

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

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.
- URLs are read from a URLfrontier 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.

- 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. \Rightarrow PageRank

A matrix $\begin{cases} \frac{1}{|V|} P_{ij} & \text{if } i, j \in V \\ 0 & \text{otherwise} \end{cases}$ is called a "stochastic" matrix. \Rightarrow always have a principal left vector

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 $M_{ij} = \begin{cases} 1 & \text{if } i \text{ links to } j \\ 0 & \text{otherwise} \end{cases}$
- ③ if row is all zero set all elements to $\frac{1}{N}$
- ④ divide 1s by the # 1s in row
- ⑤ multiply M by $(1 - \alpha)$
- ⑥ add $\alpha \cdot \frac{1}{N}$ to every element

α : teleport probability.

M is ergodic if $\exists T_0 > 0$: if at $t=0$ we are at " i ", at $t \geq T_0$ $P(i, j) \neq 0$ for all $j \geq 0$

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:

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

$\vec{\pi} = \lim_{t \rightarrow \infty} \vec{x}_0 P^t \Rightarrow$ 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 \Rightarrow hard to implement.

- Pages for informational queries
- | | | |
|-------------------------------------|----|--|
| ① authorities in the topic of query | or | ③ crafted pages that are hub of links to authority pages |
|-------------------------------------|----|--|

We give each page two scores:

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

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

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: $\begin{cases} \vec{h} \leftarrow AA^T\vec{h} \\ \vec{a} \leftarrow A^TA\vec{a} \end{cases}$
which is extremely similar to eigenvalues:

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

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

Hypertext-induced Topic Search

or HITS is the 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.

Operating



System
Concepts

Second Edition

Abraham Silberschatz
Peter Baer Galvin

Greg Gagne

Peter Baer Galvin

Greg Gagne