Behavioral Data Mining

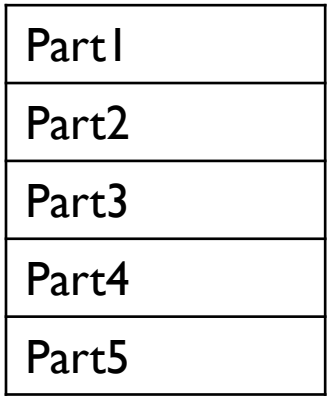
Lecture 4
Dealing with Text

Cross-Validation (from last time)

Is a method for simulating the behavior of a model on new data, and estimating its performance.

- Allows tuning of parameters like priors, model d.o.f.
- Can be used to avoid or minimize over-fitting.

K-fold cross-validation partitions data into k equal-sized subsets.



Cross-Validation (from last time)

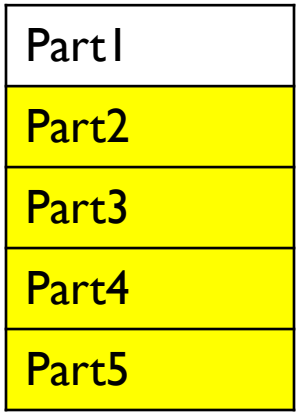
Is a method for simulating the behavior of a model on new data, and estimating its performance.

- Allows tuning of parameters like priors, model d.o.f.
- Can be used to avoid or minimize over-fitting.

K-fold cross-validation partitions data into k equal-sized subsets.

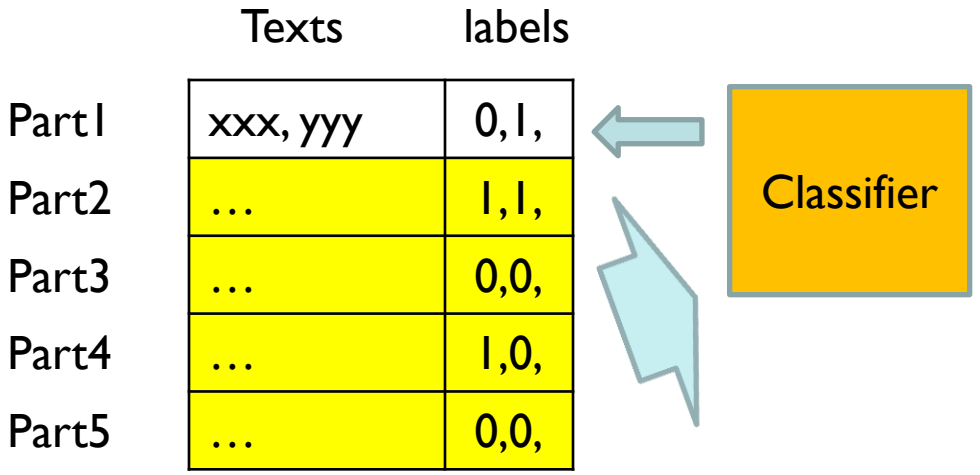
One partition is withheld as test data.

A model is built using the other partitions.



Cross-Validation

The model is evaluated on the remaining partition, (e.g. predict labels from text, compute F_1 scores).



The process can be repeated with other partitions, i.e. hold out Part2, build model from other partitions, test on Part2,…

Model Tuning

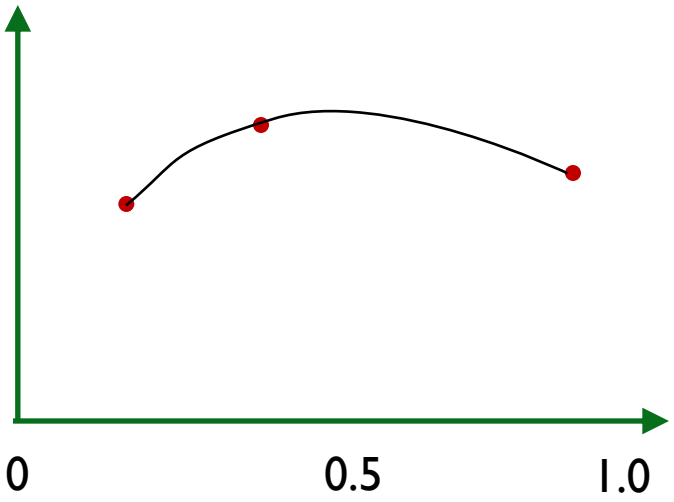
Cross-validation is an extremely useful tool for tuning models: setting priors, smoothing constants, and dimensions.

The optimal (e.g. maximum likelihood) values for these parameters are almost never optimal for predicting new data.

Parameters selected only using likelihoods on training data model both systematic and random effects, i.e. effects particular to the given training dataset. That is they over-fit the training data.

Running the model on new data allows overfit to be measured, and by iterative search, the best model parameters can be found.

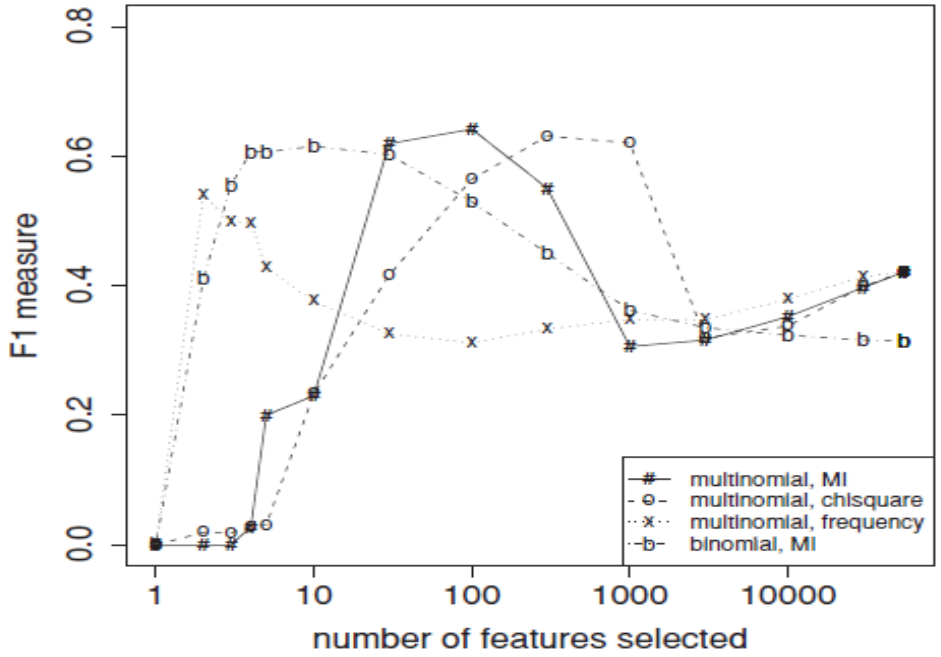
Model Tuning



e.g. Cross-validation accuracy as a function of the α-smoothing parameter, interpolated from 3 trial values (model/eval. passes).

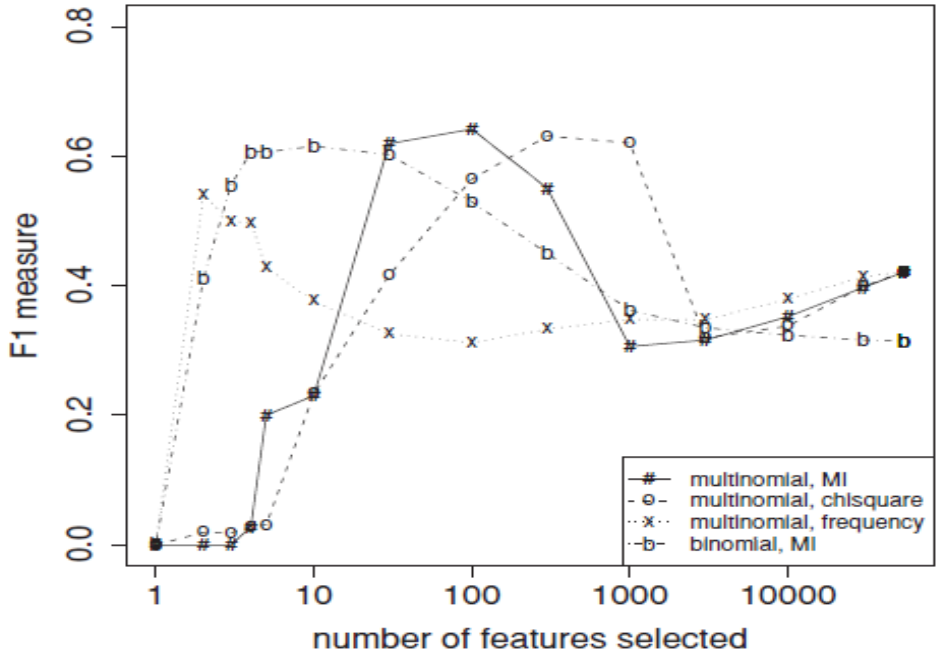
We can tune a small number of parameters this way.

Model Tuning



You can try a local optimization procedure (e.g. Newton's method), although the parameter dependence can be complex and the score may have multiple maxima. If you want to automate this, see e.g. BBOB-2012 (Black-Box Optimization Workshop).

Online Model Tuning



If using an automated procedure, you can continue optimizing even after the model is running online – this will compensate for any biases in the offline training dataset.

Text Search

We want to be able to issue queries like:

- "java" – a single word query
- "java beans" – a phrase query
- "java" AND "coffee" – a boolean query
- Near("java", "coffee", 2, "ordered") – a proximity query

We may also want to generalize the query to improve recall:

- Using stemming "farming" → "farm"
- Case insensitive search
- Ignore punctuation

Inverted Index

Is a mapping from terms to documents containing them, and possibly the positions of terms in those documents.

1. Get the latest *Java* Software and explore…
2. *Java Beans* are reusable software components for Java…
3. *Java coffee* is a coffee produced on the island of Java…

Term	(doc, pos), …. this is the "posting list"
java	(1, 4), (2, 1), (2, 8), (3, 1), (3, 11)
software	(1, 5), (2, 5)
bean(s)	(2, 2)
coffee	(3, 2)

Text Search

We can answer these queries by merging the posting lists for each word in the query:

- "java" – a single word query
- "java beans" – a phrase query
- "java" AND "coffee" – a boolean query
- Near("java", "coffee", 2, "ordered") – a proximity query

Text Search

We can answer these queries by merging the posting lists for each word in the query:

- "Java" – a single word query
- "Java beans" – a phrase query
- "Java" AND "coffee" – a boolean query
- Near("Java", "coffee", 2) – a proximity query

Q: How big is the index as just described (in terms of n = #docs and m = avg doc length)?

Text Search

We can answer these queries by merging the posting lists for each word in the query:

- "Java" – a single word query
- "Java beans" – a phrase query
- "Java" AND "coffee" – a boolean query
- Near("Java", "coffee", 2) – a proximity query

Q: How big is the index as just described (in terms of n = #docs and m = avg doc length)?

Q: How big is the index without word positions (and hence duplicate terms in each doc)?

Text Search

We can answer these queries by merging the posting lists for each word in the query:

- "Java" – a single word query
- "Java beans" – a phrase query
- "Java" AND "coffee" – a boolean query
- Near("Java", "coffee", 2) – a proximity query

Q: How big is the index as just described (in terms of n = #docs and m = avg doc length)? nm = total corpus length

Q: How big is the index without word positions (and hence duplicate terms in each doc)? ≈ nm/log m

Text Search

With this approach, which of these queries would take longer to generate a given number of hits?

- "Java" – a single word query
- "Java beans" – a phrase query

Phrase Indexing

With this approach, which of these queries would take longer to generate a given number of hits?

- "Java" – a single word query *
- "Java beans" – a phrase query *****

Suppose instead we directly indexed all the two-word phrases, with positions.

How big (in n and m) would such an index be?

Phrase Indexing

With this approach, which of these queries would take longer to generate a given number of hits?

- "Java" – a single word query *
- "Java beans" – a phrase query *

Suppose instead we directly indexed all the two-word phrases, with positions. How long would the searches take?

How big (in n and m) would such an index be? = nm

It's a reasonable cost, space permitting, since phrase queries are very common.

Three- and four-word phrase indexes have the same size, although the two-word phrase index accelerates those anyway. e.g. "zero sum game"

Bag-of-words and n-grams

So far in search and with the NB classifier, we treated documents as unordered multisets of words, or as counts over a dictionary. This is a bag-of-words model.

B.O.W. has been shown to work well for many text modeling tasks, but has some serious limitations. e.g. consider these sequences in movie reviews:

"good"

"very good"

"not bad"

"it's da bomb"

"it's a bomb"

Bag-of-words and n-grams

N-grams are consecutive n-tuples of words occurring in the documents, i.e. “phrases” as we just discussed.

N-grams can have very different meaning from their constituent words, and modeling them can considerably improve linguistic algorithms. Arguably they are richer than “logical” equivalents:

“no good”

“not great”

“not bad”

“bad”

Including these tuples into the model allows the system to learn appropriate weights for each.

Bag-of-words and n-grams

n-gram models provide progressively more convincing models of language as n increases. See e.g.

https://www.ai-class.com/course/video/videolecture/226

and the following segments…

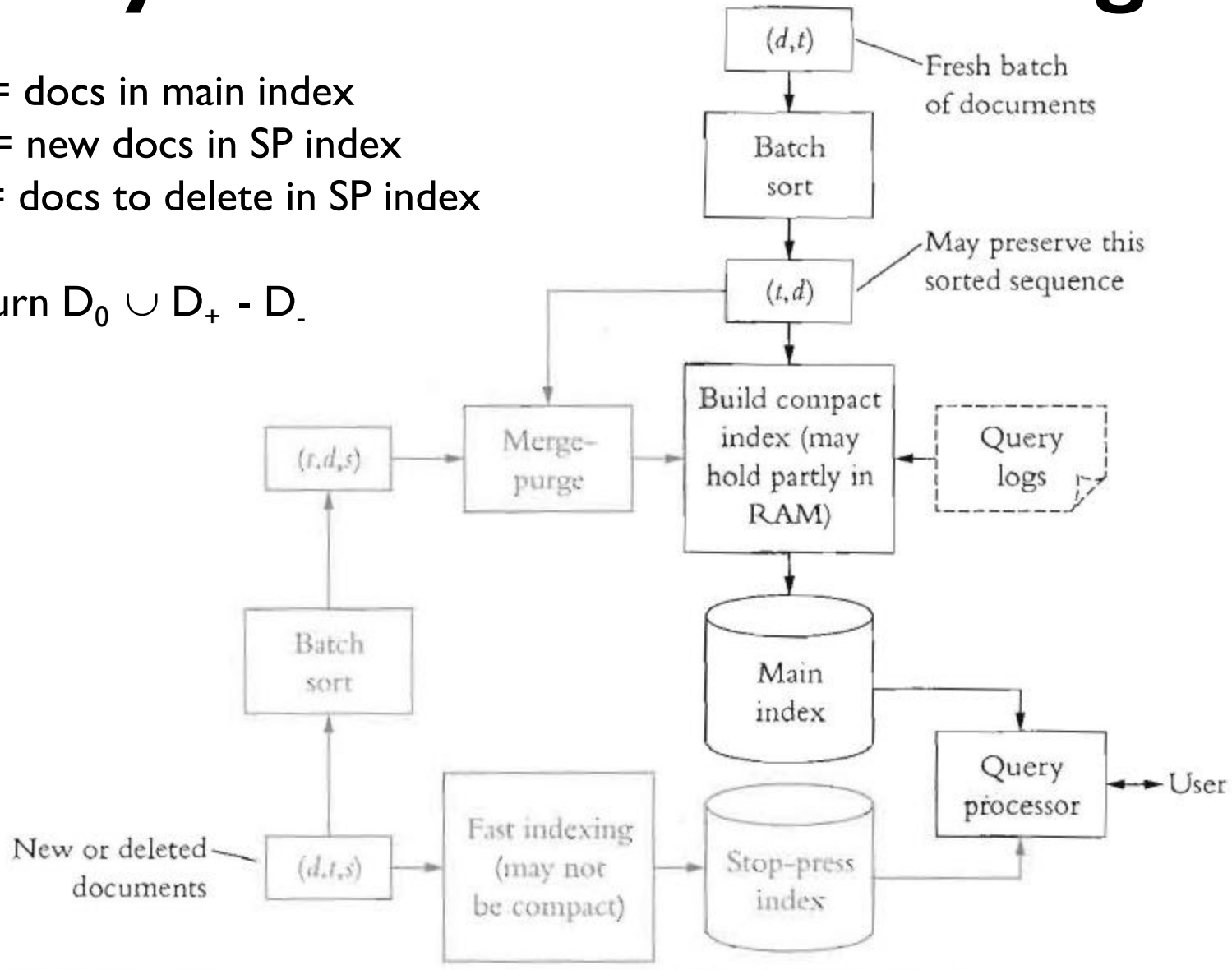
The dependence between n-grams and their sub-sequences confounds some algorithms, e.g. naïve Bayes will give biased results on docs with salient n-grams.

But other algorithms (regression or SVMs) model n-grams as increments over dependent n>k-grams, and avoid this bias.

Dynamic Batch Indexing

D_0 = docs in main index
D_+ = new docs in SP index
D_- = docs to delete in SP index

Return $D_0 \cup D_+ - D_-$



From Chakrabarti "Mining the Web"

Tokenizing

The indexer needs discrete objects (usually words) to work with, and a compact representation of them. The process of extracting words from a text is called tokenizing.

Each token is normally represented with an integer, either by hashing or by using a dictionary (hashmap(string) → int).

If direct hashing, the index set size should be > $nterms^2$

The dictionary has the advantage that the set of integers it maps to is normally dense.

The dictionary index of a word can be its descending frequency rank. This can have performance advantages when doing matrix operations on the data.

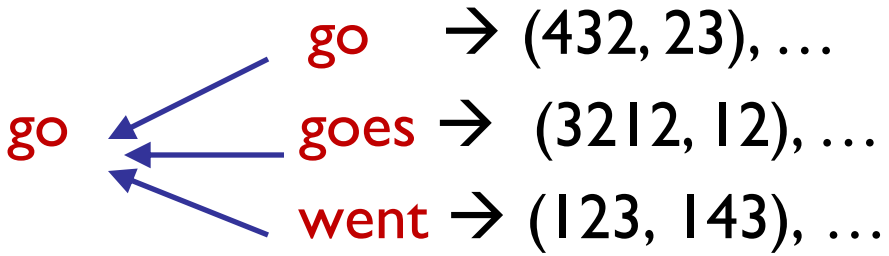
Stemming

- Morphological (e.g. Porter stemmer) goodly → good
- Lemmatization (e.g. using WordNet) better → good

Commercial search engines may also use Porter + exception rules.

Luckily the most irregular forms, e.g. irregular verbs, are the most common was → be, went → go etc. So short lists of exceptions can provide good coverage.

You can't do unstemmed searches with a stemmed index, but you can do stemmed searches with an unstemmed index:



Ranking and the Vector Space Model

For ranked search we want the documents that best match the query. Intuitively, those could be the docs in which the query terms occur most frequently:

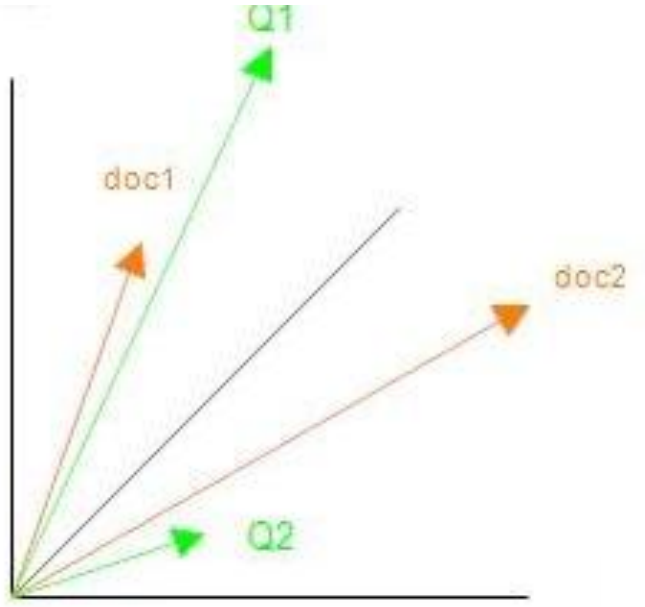
$$score_j = \frac{\sum q_i d_{ji}}{|d_j|}$$

Where q_i is the count of a term i in the query (0 or 1), and c_{ij} is the count of term i in doc j.

This expression is a scaled version of the inner product between q and d_j treated as vectors. The dimensions of this space correspond to the terms in the dictionary.

Ranking and the Vector Space Model

The inner product is the cosine between query and document vectors in the high-dimensional term space.



TF-IDF

This simple formula doesn't work well. Its dominated by the most common terms.

e.g. if you searched for "local election," hits will be dominated by the more common word "local," but "election" is much more relevant to what you want.

TF-IDF stands for Term Frequency – Inverse Document Frequency, and captures the importance of terms.

TF(d,t) = term frequency, the number of times the term t occurs in the document d

IDF(t) = inverse document frequency, one common form of which is

$$IDF(t) = \log\left(\frac{1+|D|}{|D_t|}\right)$$

where D_t is set of docs containing t

TF-IDF

Then TF-IDF(d, t) = TF(d, t) IDF(t)

This defines the weight of term t in the representation of a document. This modified vector inhabits the term space, and is again compared with a query via the inner product.

Because of linearity of the inner product, each posting list can be sorted in descending order of TF-IDF(d, t), and then single-word search hits will be returned in decreasing score order.

Multi-term queries

Search engines fall back on heuristics to do fast multi-term search.

They typically start by ranking the query terms by IDF(t), highest first. Then they truncate the posting lists at a length or score threshold:

Election: d1 > d2 > d3 > d4 > d5 > d6 > …

Local: d7 > d8 > d2 > d9 > d1 > d10 > …

The final doc ordering is the merge of the two list orders, with shared documents re-scored by summing, and re-inserted.

Probabilistic IR

TF-IDF is intuitive, but ad-hoc. There are many probabilistic models of search and retrieval. We can take a Bayesian approach and try to maximize:

$$P(d|q) = \frac{P(q|d)P(d)}{P(q)}$$

Where d is a document, q is the query. P(q) is constant for a given query, and usually we can set all P(d) to the same prior.

Probabilistic IR

A simple model for P(q|d) is

$$\hat{p}(q \mid d) = \prod_{t \in q} \hat{p}(t \mid d) = \prod_{t \in q} \frac{tf(t,d)}{|d|}$$

Where: tf(t, d) = frequency of term t in doc d

|d| = length of doc d

And taking logs we have

$$\log(\hat{p}(q \mid d)) = \sum_{t \in q} \log\left(\frac{tf(t,d)}{|d|}\right)$$

Which fits the vector space template.

Probabilistic IR

Issues: novel query terms – the model assigns zero probability to the document – too conservative.

Same problem as naïve Bayes classifier with new terms, and same solutions work: smoothing or mixtures. E.g.

$$\hat{p}_{sm}(q \mid d) = \lambda \hat{p}(q \mid d) + (1-\lambda)\hat{p}(q)$$

Where $0 < \lambda < 1$, and p(q) is the probability of the term occurring in the corpus.

For completely novel terms (not in the corpus) can use a "count" of 0.5.

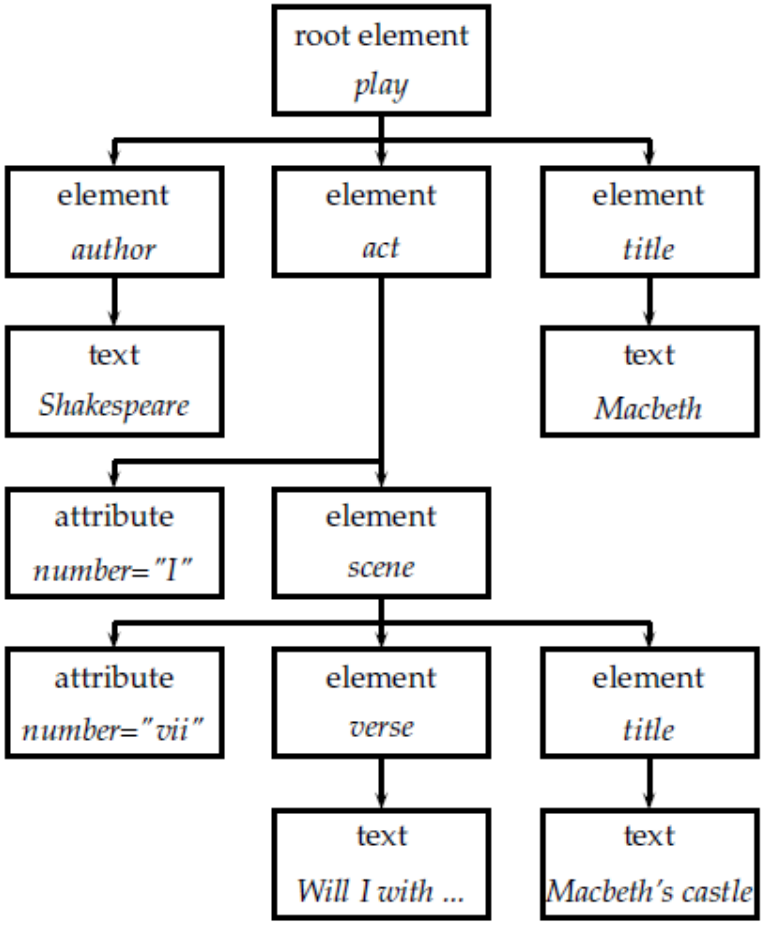
There are better models that query generation, e.g. Zhai and Lafferty's "risk minimization" 2001.

Probabilistic IR

	tf.idf	LM	%chg	I/D	Sign	Wilc.
Rel:	10485	10485				
Rret.:	5818	6105	+4.93	32/42	0.0005⋆	0.0003⋆
Prec.						
0.00	0.7274	0.7805	+7.3	10/22	0.7383	0.2961
0.10	0.4861	0.5002	+2.9	26/44	0.1456	0.1017
0.20	0.3898	0.4088	+4.9	24/45	0.3830	0.1405
0.30	0.3352	0.3626	+8.2	28/47	0.1215	0.0277⋆
0.40	0.2826	0.3064	+8.4	25/45	0.2757	0.0286⋆
0.50	0.2163	0.2512	+16.2	26/40	0.0403⋆	0.0007⋆
0.60	0.1561	0.1798	+15.2	20/30	0.0494⋆	0.0025⋆
0.70	0.0913	0.1109	+21.5	14/22	0.1431	0.0288⋆
0.80	0.0510	0.0529	+3.7	8/13	0.2905	0.2108
0.90	0.0179	0.0152	−14.9	1/4	0.3125	undef
1.00	0.0005	0.0004	−11.9	1/2	0.7500	undef
Avg:	0.2286	0.2486	+8.74	32/50	0.0325⋆	0.0015⋆
Prec.						
5	0.5320	0.5960	+12.0	15/21	0.0392⋆	0.0125⋆
10	0.5080	0.5260	+3.5	14/30	0.7077	0.1938
15	0.4933	0.5053	+2.4	14/28	0.5747	0.3002
20	0.4670	0.4890	+4.7	16/34	0.6962	0.1260
30	0.4293	0.4593	+7.0	20/32	0.1077	0.0095⋆
100	0.3344	0.3562	+6.5	29/45	0.0362⋆	0.0076⋆
200	0.2670	0.2852	+6.8	29/44	0.0244⋆	0.0009⋆
500	0.1797	0.1881	+4.7	30/42	0.0040⋆	0.0011⋆
1000	0.1164	0.1221	+4.9	32/42	0.0005⋆	0.0003⋆
RPr	0.2836	0.3013	+6.24	30/43	0.0069⋆	0.0052⋆

XML retrieval

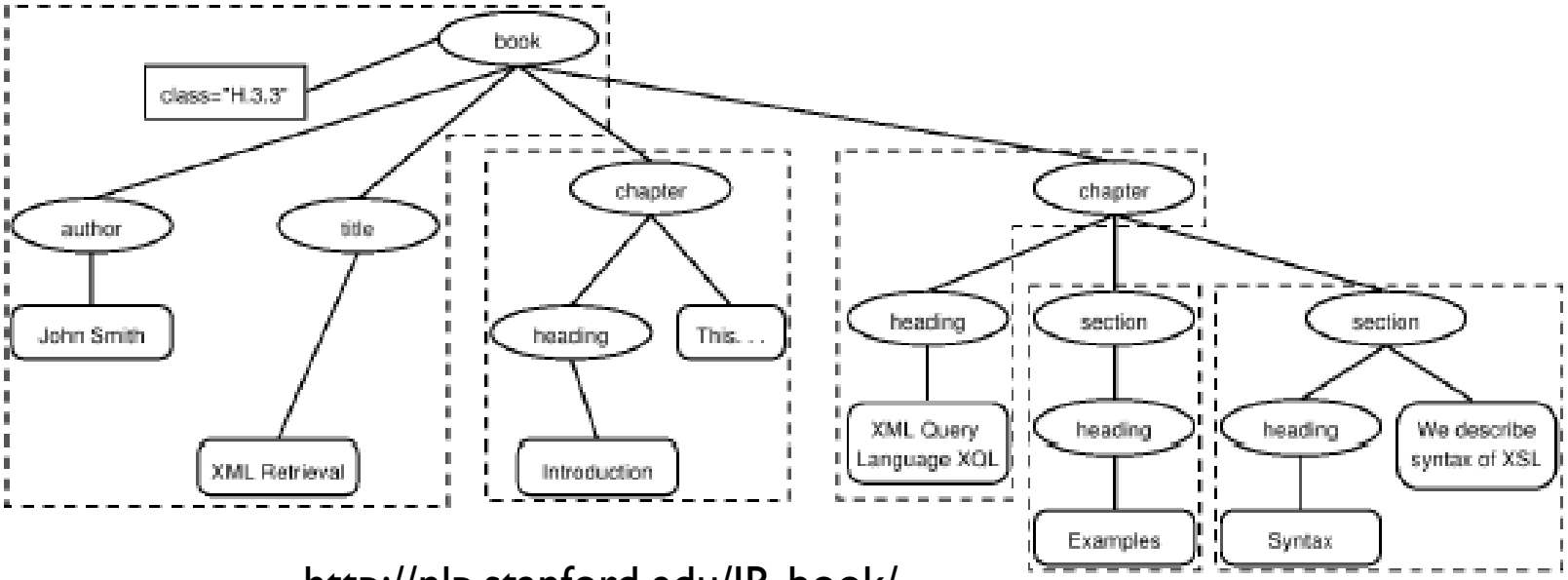
XML data is naturally hierarchical, and there may or may not be a natural notion of "document". What should we index?



http://nlp.stanford.edu/IR-book/

XML retrieval

Typically "docs" are created from the content (text nodes) of designated nodes down to designated children (fragments in ML)



http://nlp.stanford.edu/IR-book/

Of course, the data returned must be valid XML, so the children of retrieved intermediate nodes need to be scanned as well.

Summary

X-validation: useful for quantifying performance and also for tuning a few critical model parameters.

Text search uses inverted indices, indices can be tailored to the types of queries to be performed.

Stemming, lemmatization, case/punctuation normalization improve recall.

Vector space model, TF-IDF and fast score-sorted retrieval.

Probabilistic IR.

XML indexing/retrieval (preview)