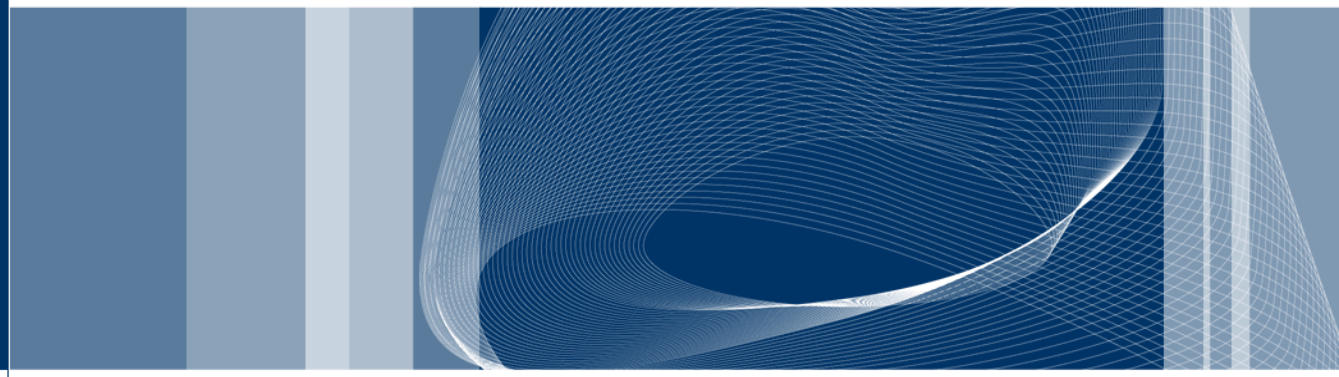
POLITECNICO DI MILANO

Natural Language Processing



Text Search

Natural Language Processing

Some slide content based on textbooks:
Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition *by Daniel Jurafsky and James H. Martin*
An Introduction to Information Retrieval by Christopher D. Manning, Prabhakar Raghavan, & Hinrich Schütze

Mark Carman

Web Search

Basics of Web Search Engines

Web Search and Information Retrieval

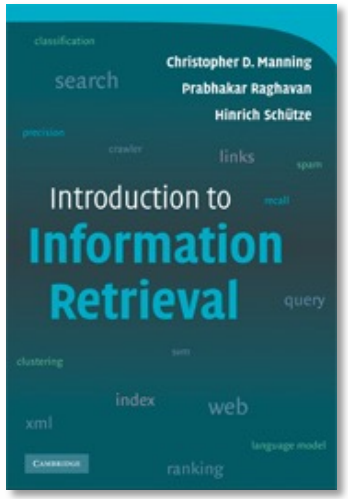
- area of research which concerns itself with technology for finding information
- makes use of technology from Machine Learning/Data Mining, Natural Language Processing (NLP), and Human Computer Interaction (HCI)

Today's textbook:

An Introduction to Information Retrieval

by Christopher D. Manning, Prabhakar Raghavan, & Hinrich Schütze

- Free online: https://nlp.stanford.edu/IR-book/information-retrieval-book.html



Contents

Source https://www.pexels.com/photo/google-internet-online-search-search-48123/

- What is Information Retrieval?
- Term weighting
- Building Indices
- Preprocessing
- Crawling

Information Retrieval

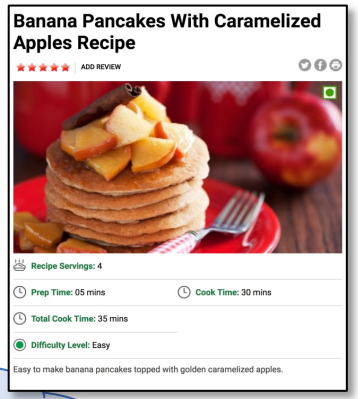
What is Information Retrieval?

Task of finding content: text, images, video, etc.

- that is useful (i.e. relevant) to user's **information need**

Source: https://www.bbc.com/news/world-us-canada-38682574

My inauguration crowd was the biggest ever. Period! Somebody find me some pictures



Source: https://food.ndtv.com/recipe-banana-pancakes-with-caramelized-apples-588999

Hmm. Those banana and apple pancakes look so good! I wonder how you make them …

Source: https://www.flickr.com/photos/gageskidmore/29381357345

My fingers look pretty big to me. They can't be shorter than the average, can they?

Gender	Right hand (RH) (mm)			Left hand (LH) (mm)		
	Rt2D	Rt3D	Rt4D	Lt2D	Lt3D	Lt4D
Male	74.278	80.172	75.715	74.61	80.421	75.697
Female	65.894	71.807	67.541	66.192	71.987	67.344

Source: https://www.researchgate.net/publication/281750703_Digit_ratio_2D4D_Index_finger_Ring_finger_in_the_right_and_left_hand_of_males_and_females_in_Malaysia

Text retrieval isn't that hard

We just turn **information needs** into **text queries**

- and look for documents containing those keywords

Source: https://www.flickr.com/photos/gageskidmore/29381357345

Hmm. Those banana and apple pancakes look so good! I wonder how you make them …

large collection containing many **billions** of documents:

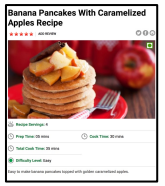
millions of docs containing word "banana"

thousands of docs containing words: banana + apple

hundreds of docs containing words: banana + apple + pancake

tens of docs containing words: banana + apple + pancake + make

relevant documents



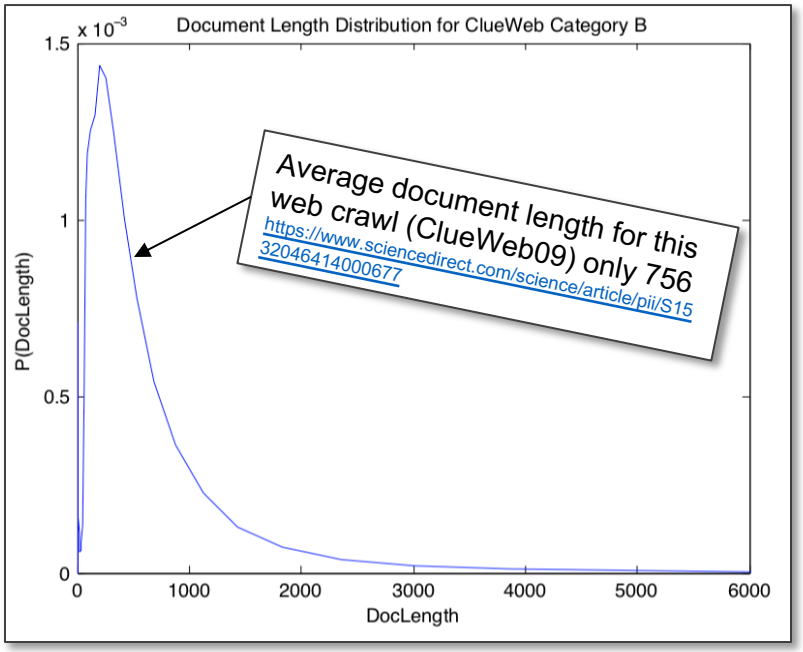
So web search isn't hard?

Typical vocabulary of adult in range 20,000 to 35,000 words:

- See: https://www.economist.com/johnson/2013/05/29/lexical-facts

Typical web document has **length** that is **much smaller** than 20k

- consequently each document's vocabulary is also much smaller than 20k
- and importantly: vocabulary is well distributed over different documents



Heavy lifting in text retrieval is done by **vocabulary matching**!

- much of web search is about making **fast indexes** for finding documents containing keywords

Hold on, its' not quite that simple …

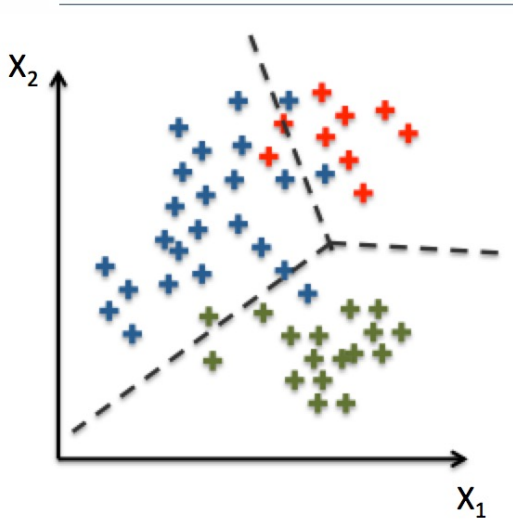
What if **no document** contains all of the query keywords?

- or if **many documents** contain all of the query terms?

Possible answers:

1. Assign a score to the **importance** of different keywords
   - some keywords are more **discriminative** than others, so weight them up
2. Expand **document representation** to include more information
   - e.g. include AnchorText from incoming links → very important in Web Search!
3. Train a **Machine Learning** algorithm?

Is retrieval just text classification?



Why not just train a ML classifier?

- input features: <*query terms, document content*>
- output: ***Probability*** user finds document relevant

Could **NOT** be a linear classifier

- since interaction between query terms and document terms would NOT be taken into account

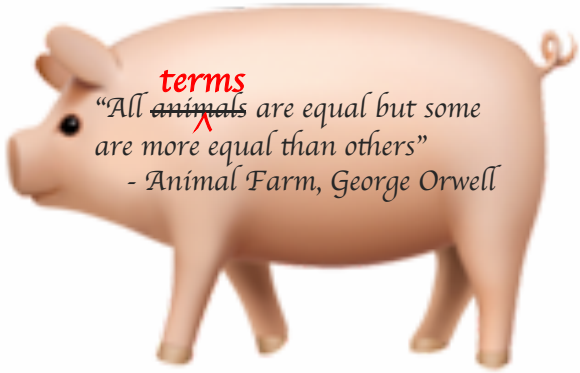
Could train a simple quadratic model:

- including **all pairwise interactions** between query and document terms
- **problem**: on a 'small' vocabulary of 10^5 there would be 10^{10} interactions
  - that's a LOT of parameters: 10 billion
  - would need a LOT of training data: <query, document> pairs + labels

We will return to idea of treating retrieval as a classification/regression problem later

Term Weighting

Term weighting



All terms carry meaning, but some terms are **more discriminative** than others

- making them **more important** than others
- when searching for documents we need to **weight up** the important terms

Source: https://www.publicdomainpictures.net/en/view-image.php?image=240887&picture=weights

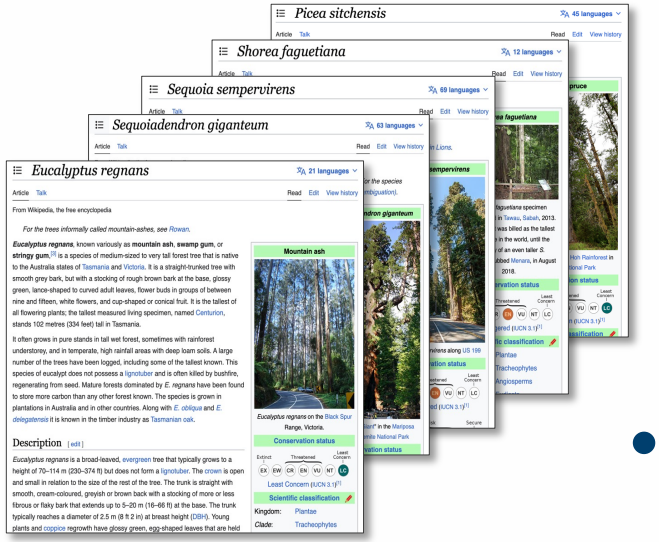
- now discuss important heuristics for ranking documents based on query terms they contain
  - Inverse Document Frequency (IDF)
  - TF-IDF
  - Cosine similarity
  - BM25

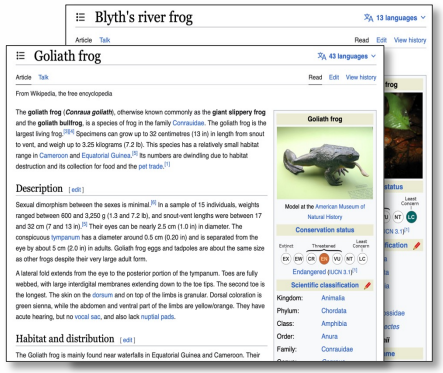
Term weighting - example

Source: https://en.wikipedia.org/wiki/Agalychnis_callidryas#/media/File:Red-eyed_Tree_Frog_(Agalychnis_callidryas)_1.png

Imagine you're searching for name of a cool frog you saw in forest

- you enter the query: **giant tree frogs**
  - but **no document** in Wikipedia contains all three keywords 😟
- so you try dropping one of the keywords from the query:
  - many documents containing: **giant+tree**,
  - a couple containing: **giant+frog**
  - and a few for: **tree+frog**





- which set is most useful?
  - in general, the **smaller** the set, the more **on topic** it is
  - and more likely it is to be useful …

Motivating IDF

Source: https://en.wikipedia.org/wiki/Uncle_Sam#/media/File:J._M._Flagg,_I_Want_You_for_U.S._Army_poster_(1917).jpg

To find and rank documents for query:

- could run search with all possible subsets of query terms
  - and rank results by how few documents are returned
  - but with k query terms, might end up running 2^k searches
  - so lots of searches 😨 and wasted computation
- is there a faster and more principled way to do this?

Could **estimate** how many documents would likely be returned for given subset

- by multiplying total **number of documents** in collection by **probability** that **random document** contains that set of query terms
  - i.e.: $\#docs(q) \cong N * P(q \in vocab \text{ of } random\ document)$

Since our aim is to rank documents, we could ignore N and **rank documents** by:

- how **unlikely** we are **to find a random document** with all those terms:
  - i.e. rank by inverse probability:
    $score(q,d) = 1 / P(q \cap d \in random\ document)$
    - where $q \cap d$ denotes set of terms in both the query q and the document d

Term weighting – IDF

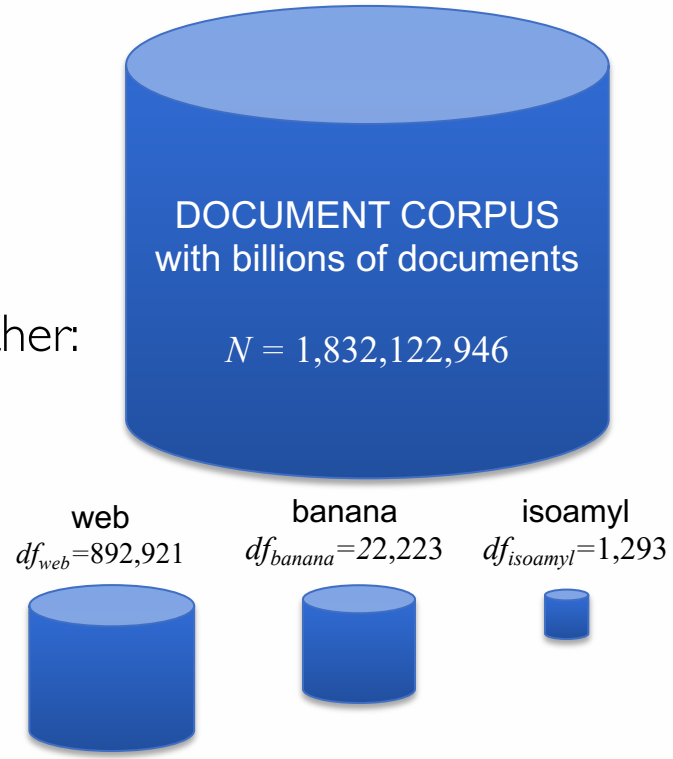
What is probability that randomly chosen document contains set of keywords?

- assuming that terms are independent of one another:

$$P(q \in d') = \prod_{t \in q} P(t \in d') = \prod_{t \in q} \frac{\mathrm{df}_t}{N}$$

N = number of documents in corpus

df_t = # docs in corpus containing term t
(a.k.a. *document frequency*)



- but we wish to rank documents by how **unlikely** they are
  - so score by the **inverse probability**: 1/P
  - and to make **score additive** (rather than multiplicative) we compute the logarithm
- so in the end, rank documents by their **negative log-likelihood** of containing so many query terms:

$$score(d) = -\log \prod_{t \in q \cap d} P(t \in d') = \sum_{t \in q \cap d} \log \frac{N}{\mathrm{df}_t}$$

Term weighting – IDF (cont.)

Returning to discussion that some terms **more discriminative** than others…

- **Inverse Document Frequency (IDF)** simply weights each term by negative log probability of finding term in random document:

$$\text{idf}_t = \log \frac{N}{\text{df}_t}$$

This is standard formula from **Information Theory:**

$$Information(event) = -log\ P(event)$$

- quantifies amount of surprise we experience at learning that the event occurred
- the more surprise, the more information we have learnt
- also used to determine how many bits to use to represent event (e.g. observing particular byte) when compressing a file

Source: https://commons.wikimedia.org/wiki/File:SURPRISE.jpg

Assumption:

- the more **surprising** the document is with respect to the query
- the more likely it is to be **relevant** to the query

Term weighting – IDF (cont.)

There are other common versions of Inverse Document Frequency (IDF)

- such as that based on the log-odds:

$$\text{idf}_t = \log \frac{[1\text{-}P(t \in rand)]}{P(t \in rand)}$$

- which results in the following document score:

$$score(d) = \sum_{t \in q} \log \frac{N - \text{df}_t + \frac{1}{2}}{\text{df}_t + \frac{1}{2}}$$

- little difference, if any, between the two formulations
  - (providing term isn't super common, so $\text{df}_t < N/2$)
  - for most terms document frequency is small compared to size of collection: $N - \text{df}_t \cong N$
- also a small amount of smoothing has also been applied
  - 0.5 has been added to the counts so that terms with very small document frequencies (1, 2, etc.) don't dominate the ranking

Term Weighting – TF-IDF

Term weighting – TF-IDF

Isolated-term correction would fail to correct typographical errors such as flew form Heathrow, where all three query terms are correctly spelled. When a phrase such as this retrieves few documents, a search engine may like to offer the corrected query flew from Heathrow. The simplest way to do this is to enumerate corrections of each of the three query terms (using the methods leading up to Section 3.3.4) even though each query term is correctly spelled, then try substitutions of each correction in the phrase. For the example flew form Heathrow, we enumerate such phrases as fled form Heathrow and flew fore Heathrow. For each such substitute phrase, the search engine runs the query and determines the number of matching results.

IDF weights vocabulary terms

- but there is **more information** in a document than just its **vocabulary**!

- some documents contain the **same query term many times**
  - likely to be discussing the "query topic" many times
  - all else being equal, they are more likely to be relevant to the query

- simplest option to include this **term frequency** information is to directly weight the score by it:

$$score(q,d) = \sum_{t \in q} \mathrm{tf}_{t,d} \log \frac{N}{\mathrm{df}_t}$$

  - where $\mathrm{tf}_{t,d}$ = # of occurrences of term t in document d (a.k.a. *term frequency*)

- hold on, but where did that formula come from?
  - where is the justification for just inserting the term counts?
  - I'm glad you asked! 😉

Motivating TF-IDF

Source: https://en.wikipedia.org/wiki/Uncle_Sam#/media/File:J._M._Flagg,_I_Want_You_for_U.S._Army_poster_(1917).jpg

Instead of just calculating:

P(random document contains term)

as we did for the IDF case, why not calculate:

P(random document contains term k times)

Estimation:

- again making Naïve Bayes assumption that tokens occur independently of one another, we can estimate the above by computing:

  $P(\text{random doc contains term } k \text{ times}) \cong P(\text{next token is term})^k$

- to estimate chance that next token is term, ignore document boundaries and count frequency of term over whole collection:

  $P(\text{next token is term } t) = \text{ctf}_t / \sum_{t'} \text{ctf}_{t'}$

  where ctf_t = occurrences of t in collection and $\sum_{t'} \text{ctf}_{t'}$ = length of collection (in tokens)

- and thus we have:

  $P(\text{random doc contains } tf \text{ occurrences of } t) \cong (\text{ctf}_t / \sum_{t'} \text{ctf}_{t'})^{tf}$

- computing product over query terms and taking logarithm to produce additive score:

  $score(q,d) = \sum_{t \in q} \text{tf}_{t,d} \log(\text{ctf}_t / \sum_{t'} \text{ctf}_{t'})$

- cool, but that's not quite the formula from the previous slide…
  - true, but changing the $P(t)$ estimator from using collection frequency (ctf_t) statistics to document frequency (df_t) doesn't change things drastically and may even slightly improve performance

Alternative versions of TF-IDF

Isolated-term correction would fail to correct typographical errors such as flew form Heathrow, where all three query terms are correctly spelled. When a phrase such as this retrieves few documents, a search engine may like to offer the corrected query flew from Heathrow. The simplest way to do this is to enumerate corrections of each of the three query terms (using the methods leading up to Section 3.3.4) even though each query term is correctly spelled, then try substitutions of each correction in the phrase. For the example flew form Heathrow, we enumerate such phrases as fled form Heathrow and flew fore Heathrow. For each such substitute phrase, the search engine runs the query and determines the number of matching results.

So the *term frequency – inverse document frequency* (TF-IDF) score often used to weight query terms in a document is the following:

$$score(q,d) = \sum_{t \in q} \mathrm{tf}_{t,d} \log \frac{N}{\mathrm{df}_t}$$

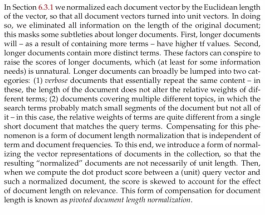
- score performs relatively well in practice
- assumes a linear relationship between term frequency and document score

- Researchers have questioned this linear assumption:
  - should, all else being equal, doubling the occurrences of a term in a document, double the score for the document?
  - probably not:
    - score should improve with increases in the term count, but not linearly
    - already expected to see term multiple times in doc after saw it the first time
  - common alternative (with little theoretical justification) is to increase score with logarithm of term count:

$$\log(1 + \mathrm{tf}_{t,d}) \quad or \quad \max(0, 1 + \log(\mathrm{tf}_{t,d}))$$

Term Weighting – Length Normalization

Term weighting – length normalisation


In Section 6.3.1 we normalized each document vector by the Euclidean length of the vector, so that all document vectors turned into unit vectors. In doing so, we eliminated all information on the length of the original document; this masks some subtleties about longer documents. First, longer documents will – as a result of containing more terms – have higher tf values. Second, longer documents contain more distinct terms. These factors can conspire to raise the scores of longer documents, which (at least for some information needs) is unnatural. Longer documents can broadly be lumped into two categories: (1) *verbose* documents that essentially repeat the same content – in these, the length of the document does not alter the relative weights of different terms; (2) documents covering multiple different topics, in which the search terms probably match small segments of the document but not all of it – in this case, the relative weights of terms are quite different from a single short document that matches the query terms. Compensating for this phenomenon is a form of document length normalization that is independent of term and document frequencies. To this end, we introduce a form of normalizing the vector representations of documents in the collection, so that the resulting "normalized" documents are not necessarily of unit length. Then, when we compute the dot product score between a (unit) query vector and such a normalized document, the score is skewed to account for the effect of document length on relevance. This form of compensation for document length is known as *pivoted document length normalization*.


In Section 6.3.1 we normalized each document vector by the Euclidean length of the vector, so that all document vectors turned into unit vectors. In doing so, we eliminated all information on the length of the original document; this masks some subtleties about longer documents. First, longer documents will – as a result of containing more terms – have higher tf values. Second, longer documents contain more distinct terms. These factors can conspire to raise the scores of longer documents, which (at least for some information needs) is unnatural. Longer documents can broadly be lumped into two categories: (1) *verbose* documents that essentially repeat the same content – in these, the length of the document does not alter the relative weights of different terms; (2) documents covering multiple different topics, in which the search terms probably match small segments of the document but not all of it – in this case, the relative weights of terms are quite different from a single short document that matches the query terms. Compensating for this phenomenon is a form of document length normalization that is independent of term and document frequencies. To this end, we introduce a form of normalizing the vector representations of documents in the collection, so that the resulting "normalized" documents are not necessarily of unit length. Then, when we compute the dot product score between a (unit) query vector and such a normalized document, the score is skewed to account for the effect of document length on relevance. This form of compensation for document length is known as *pivoted document length normalization*.

Consider a document collection together with an ensemble of queries for that collection. Suppose that we were given, for each query q and for each document d, a Boolean judgment of whether or not d is relevant to the query q; in Chapter 8 we will see how to procure such a set of relevance judgments for a query ensemble and a document collection. Given this set of relevance judgments, we may compute a *probability of relevance* as a function of document length, averaged over all queries in the ensemble. The resulting plot may look like the curve drawn in thick lines in Figure 6.16. To compute this curve, we bucket documents by length and compute the fraction of relevant documents in each bucket, then plot this fraction against the median document length of each bucket. (Thus even though the "curve" in Figure 6.16 appears to be continuous, it is in fact a histogram of discrete buckets of document length.)

On the other hand, the curve in thin lines shows what might happen with the same documents and query ensemble if we were to use relevance as prescribed by cosine normalization Equation (6.12) – thus, cosine normalization has a tendency to distort the computed relevance vis-à-vis the true relevance, at the expense of longer documents. The thin and thick curves crossover at a point p corresponding to document length ℓ_p, which we refer to as the *pivot*

In Section 6.3.1 we normalized each document vector by the Euclidean length of the vector, so that all document vectors turned into unit vectors. In doing so, we eliminated all information on the length of the original document; this masks some subtleties about longer documents. First, longer documents will – as a result of containing more terms – have higher tf values. Second, longer documents contain more distinct terms. These factors can conspire to raise the scores of longer documents, which (at least for some information needs) is unnatural. Longer documents can broadly be lumped into two categories: (1) *verbose* documents that essentially repeat the same content – in these, the length of the document does not alter the relative weights of different terms; (2) documents covering multiple different topics, in which the search terms probably match small segments of the document but not all of it – in this case, the relative weights of terms are quite different from a single short document that matches the query terms. Compensating for this phenomenon is a form of document length normalization that is independent of term and document frequencies. To this end, we introduce a form of normalizing the vector representations of documents in the collection, so that the resulting "normalized" documents are not necessarily of unit length. Then, when we compute the dot product score between a (unit) query vector and such a normalized document, the score is skewed to account for the effect of document length on relevance. This form of compensation for document length is known as *pivoted document length normalization*.

Consider a document collection together with an ensemble of queries for that collection. Suppose that we were given, for each query q and for each document d, a Boolean judgment of whether or not d is relevant to the query q; in Chapter 8 we will see how to procure such a set of relevance judgments for a query ensemble and a document collection. Given this set of relevance judgments, we may compute a *probability of relevance* as a function of document length, averaged over all queries in the ensemble. The resulting plot may look like the curve drawn in thick lines in Figure 6.16. To compute this curve, we bucket documents by length and compute the fraction of relevant documents in each bucket, then plot this fraction against the median document length of each bucket. (Thus even though the "curve" in Figure 6.16 appears to be continuous, it is in fact a histogram of discrete buckets of document length.)

On the other hand, the curve in thin lines shows what might happen with the same documents and query ensemble if we were to use relevance as prescribed by cosine normalization Equation (6.12) – thus, cosine normalization has a tendency to distort the computed relevance vis-à-vis the true relevance, at the expense of longer documents. The thin and thick curves crossover at a point p corresponding to document length ℓ_p, which we refer to as the *pivot*

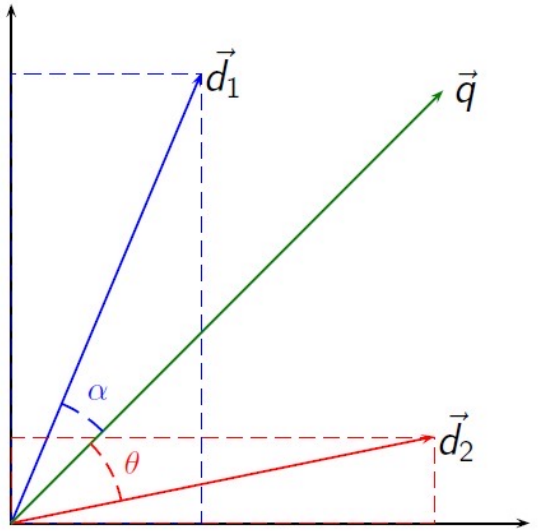
Need to normalise for the length of the document!

- longer documents have a larger vocabulary
  - more likely to contain the query terms
  - not necessarily more likely to be useful to the searcher
- shorter documents with the same count of query terms should be preferred

How to normalise for length?

- Could just divide by length of the document
  - done in many "Language Modeling" based retrieval functions:
    - E.g. http://sifaka.cs.uiuc.edu/czhai/pub/tois-smooth.pdf

Term weighting – vector space



Source: https://en.wikipedia.org/wiki/Vector_space_model#/media/File:Vector_space_model.jpg

Vector Space Model treats tf-idf values for all terms in a document as a vector representation:

$$\mathbf{d} = (\mathrm{tf}_{1,d}.\mathrm{idf}_1, ..., \mathrm{tf}_{1,d}.\mathrm{idf}_n)$$

- similarity between documents (or query and document) computed based on the angle between vectors
  - actually the cosine of angle is used instead (since it gives similarity values in range [0,1])

$$sim(\mathbf{d}_1, \mathbf{d}_2) = \frac{\mathbf{d}_1 \cdot \mathbf{d}_2}{||\mathbf{d}_1||||\mathbf{d}_2||}$$

  - using cosine similarity involves normalising by the Euclidean length of the vectors (rather than the length of the document)

$$||\mathbf{x}|| = \sqrt{\sum_{i=1}^{n} x_i^2}$$

Term weighting – pivoted length norm.

Many studies over the years on other types of normalization, such as:

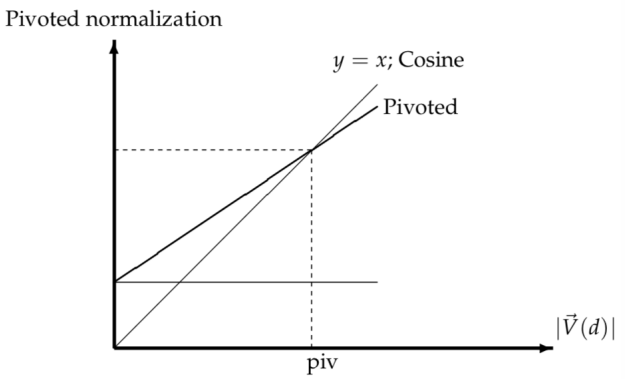
- Pivoted Length Normalisation (PLN)
  - **parameterized** form of normalisation developed specifically for retrieval
  - idea: longer documents contain more information than shorter ones
    - so could be more useful to searcher, but normalising loses length information
    - instead parameterise L_1 normalisation around average document length:

$$\frac{\mathrm{tf}_{t,d}}{L_d} \quad \rightarrow \quad \frac{\mathrm{tf}_{t,d}}{bL_d + (1-b)L_{ave}}$$

where:

$$L_d = \sum_t \mathrm{tf}_{tid} \qquad L_{ave} = \frac{1}{N}\sum_d L_d$$

$$0 \leq b \leq 1$$



Term Weighting – BM25

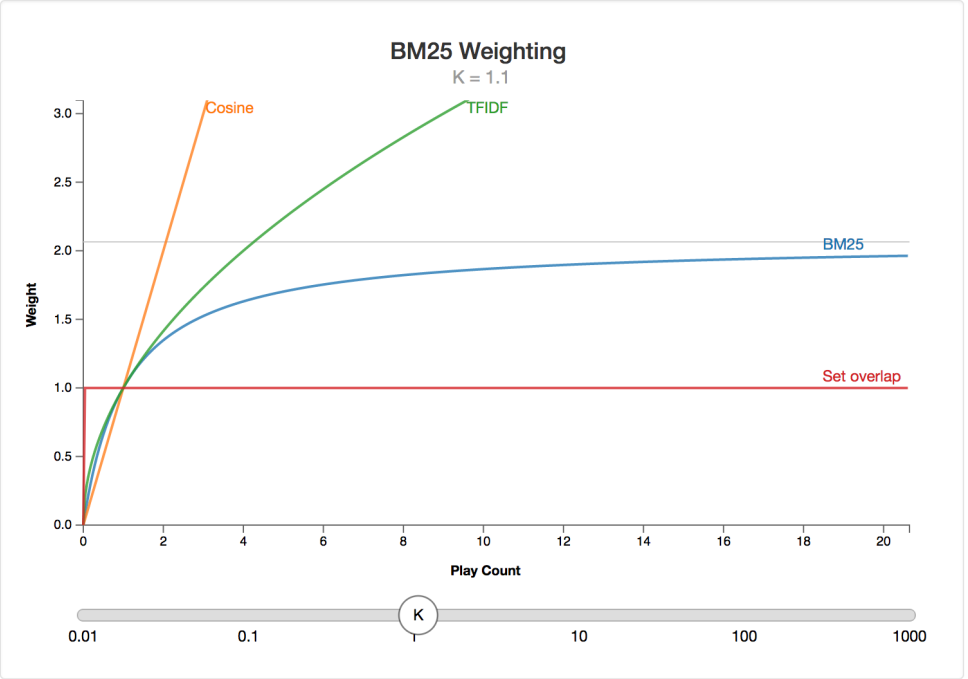
Pivoted length normalization leads to the venerable (it's been around a while) **Okapi BM25 Formula** for ranking documents:

$$RSV_d = \sum_{t \in q} \log \left[\frac{N}{\mathrm{df}_t} \right] \cdot \frac{(k_1 + 1)\mathrm{tf}_{td}}{k_1((1-b) + b \times (L_d / L_{\mathrm{ave}})) + \mathrm{tf}_{td}}$$

k_1 , b = *parameters to be set*

- why the weird name BM25?
  - BM stands for Best Match, and it was literally the 25th formula they tried ;-)
- is/was the **GOTO method** for **term-based** text retrieval
  - formula has stood test of time, with LOTS of competition along the way:
    - see article "Has adhoc retreival improved since 1994?" https://dl.acm.org/citation.cfm?id=1572081

BM25 Properties



Source: https://www.benfrederickson.com/distance-metrics/

BM25 has nice properties:

- Term importance asymptotes to maximum value
- So document containing one query term a massive number of times won't always rise to the top of the ranking

Parameters control dependence on document length

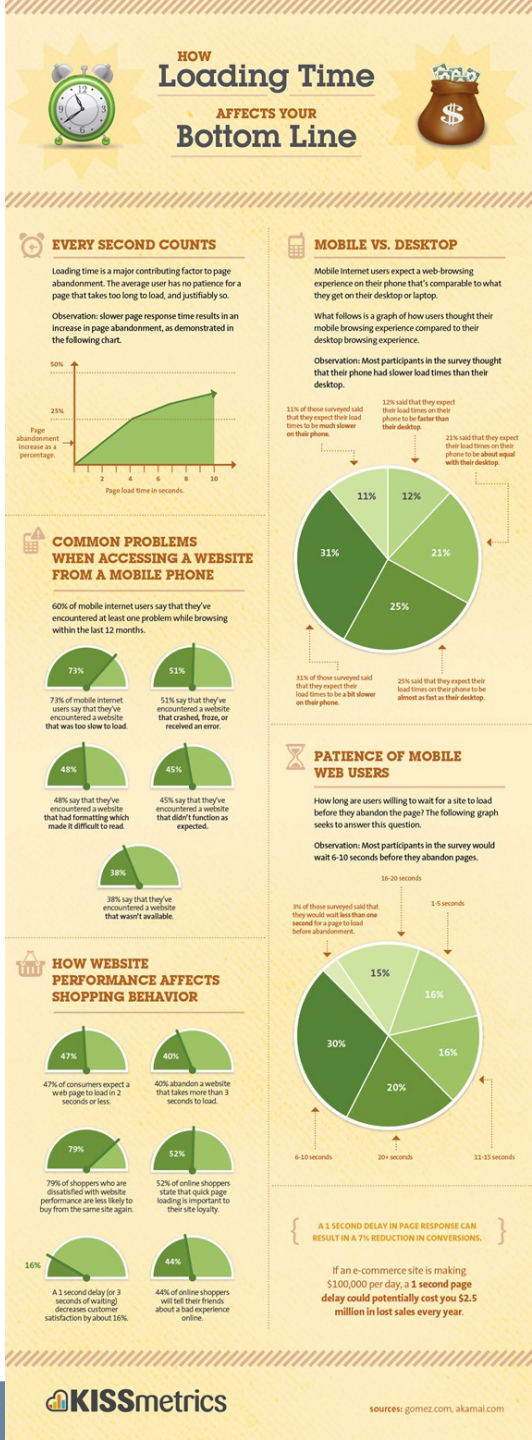
- (pivoted document length normalisation)
- Default values for parameters exist (k_1 between 1.2 & 2, b=.75), but usually one sets vai trial and error

Index Structures

Under the Hood of Search Engine

Now discuss how these retrieval measures can be calculated …FAST

- on the internet:
  - time affects attention and
  - attention is money
- so search engines
  - need to respond in tenths of a second
  - engineered to be as fast as is possible



Source:
https://neilpatel.com/wp-content/uploads/2011/04/loading-time-sml.jpg

(Inverted) Indices

Source: https://commons.wikimedia.org/wiki/File:Telefonbog_ubt-1.JPG

Inverted Indices:

- the building blocks of search engines
- Made up of **Posting Lists**
  - simply lists mapping: TermIDs => DocumentIDs
- Extremely well engineered to be as fast as possible
  - Posting lists are compressed with integer compression algorithms that
    - allow for fast decompression
    - take the statistics of text into account
    - are in blocks to allow for easy reading
  - Why compress posting lists?
    - To reduce the amount of space on disk?
    - No to reduce the amount of space in memory!

Computing Joins

Source: https://commons.wikimedia.org/wiki/File:Telefonbog_ubt-1.JPG

Computing retrieval function involves:

- finding all documents containing set of terms
- i.e. computing a join over the posting lists

Entries in the posting list are sorted:

- from most important documents (e.g. highest term counts) to least
- allows for Early Termination of results list computation
  - compute bounds on scores for documents calculation
  - lower ranked documents dropped as early as possible
- Index Pruning Techniques
  - get rid of documents that would never be retrieved for a certain query

  http://engineering.nyu.edu/~suel/queryproc/

Positional Indices

Proximate terms within document are more likely to describe query topic

- extreme case is Named Entities
- meaning depends on order of terms in bigram / trigram

Most indices record locations of terms in document

- allows for computing proximity
- position in document may also be useful
  - words at start of webpage more important

Statistically significant bigram/trigrams

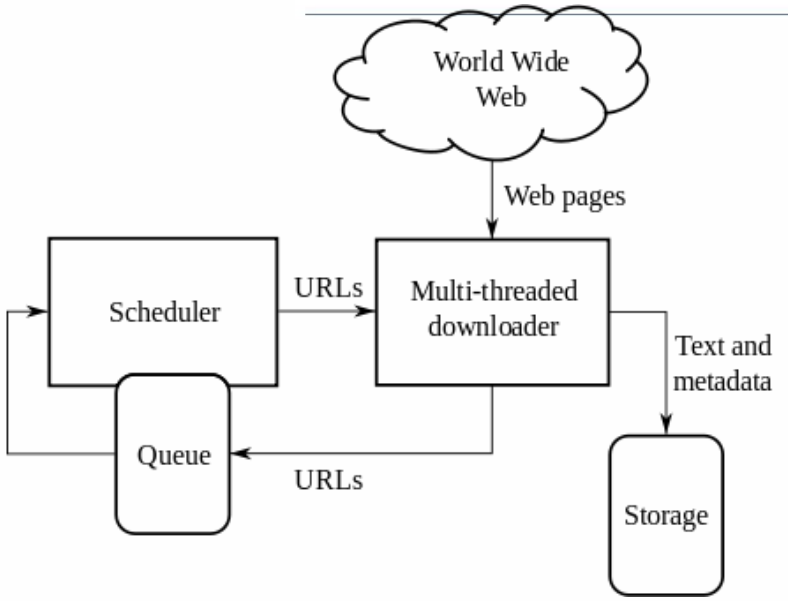
- e.g. using pointwise mutual information
- often indexed with their own posting list

Source: https://commons.wikimedia.org/wiki/File:White_House_DC.JPG

"a tree next to the white house"
vs
"the tree next to a white house"

Source: https://www.nps.gov/york/learn/historyculture/moore-house.htm

Crawlers



Source: An Introduction to Information Retrieval. Manning, Raghavan & Schütze

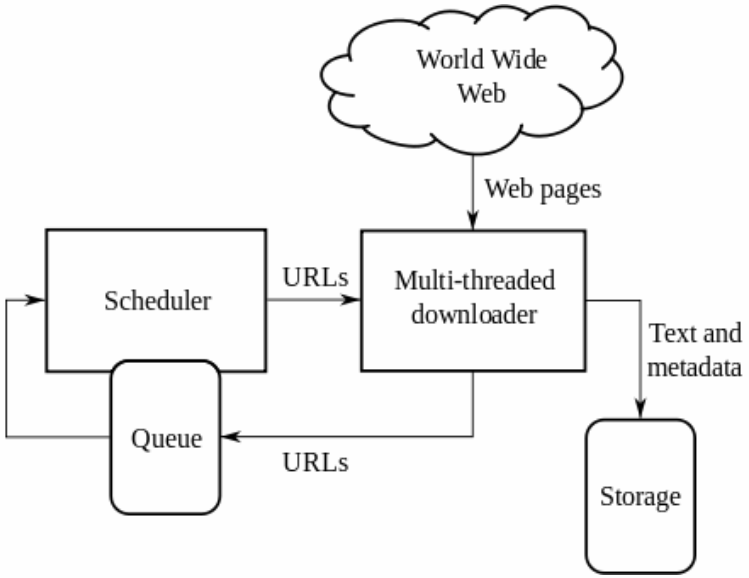
Scour web following hyperlinks in search of pages to add to index

- lots of research in past on how to crawl effectively
  - was battle ground between search engines
  - basically comes down to prioritizing URLs appropriately
  - and having LOTS of cheap computing power
- Indexes are continuously updated
- Lots of parameters that can be set/learnt, such as:
  - how frequently to re-visit a website
  - which links to prioritise, which to discard

Crawlers (cont.)

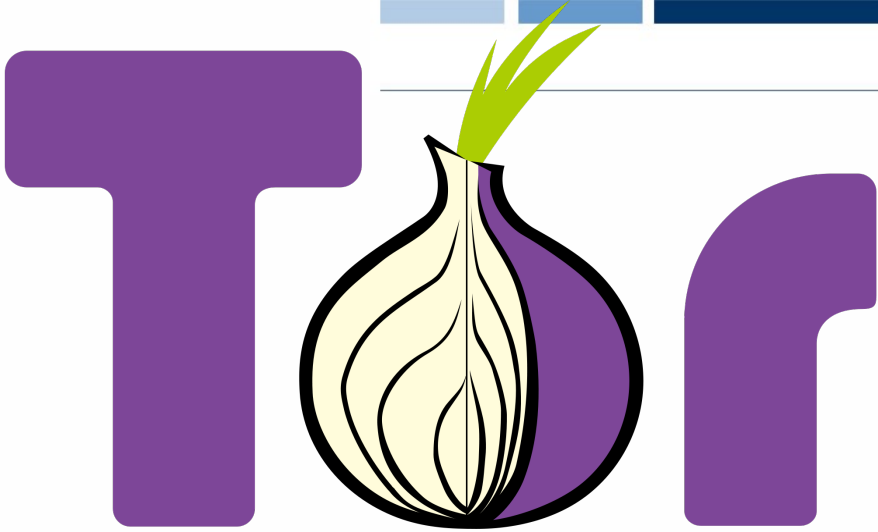
Web scale crawlers need to be robust to all types of content

- including generated content
  - used to be the case that Google had more Amazon pages than Amazon, since Amazon generated them on demand, while Google indexed them statically and kept them all in memory!
- content-based duplicate page detection
  - many URLs may map to the same content
- **distributed crawler** architecture with centralised URL list
- respect **robots.txt** files
  - text files in the root directory of a website that tell the crawler what content they can/can't crawl and any restrictions on the crawling



Source: An Introduction to Information Retrieval. Manning, Raghavan & Schütze

Crawlers - Aside



Source https://commons.wikimedia.org/wiki/File:Tor-logo-2011-flat.svg

Dark Web

- anonymous Web built around TOR (The Onion Router) gateways
  - full of all sorts of nasty content
- crawling the Dark Web is interesting because there is no DNS linking urls to IP-addresses behind TOR gateways

Further Reading

- Want to read more about that unusual BM25 formula came from?
  - Read this paper by Stephen Robertson and Hugo Zaragoza:
    - http://www.staff.city.ac.uk/~sb317/papers/foundations_bm25_review.pdf
- Want to know about the latest in compression algorithms for posting lists (inverted indices)?
  - Check out this paper by Alistair Moffat and Matthias Petri from WSDM 2018:
    - http://delivery.acm.org/10.1145/3160000/3159663/p405-moffat.pdf
- Interested in the statistics of text?
  - Have a look at Yee Whye Teh's slides on the Pitman-Yor Process:
    - http://mlg.eng.cam.ac.uk/tutorials/07/ywt.pdf

Learning to Rank

Web Search part 2: Learning to Rank

- Formulating ranking as a learning problem
- Evaluating rankings
- Learning with regression tree ensembles

Why learn to rank?

We often need to rank things:

- candidate documents in a retrieval system
- candidate products in a recommender system
- candidate answers in a question answering system
- candidate translations in a machine translation system
- candidate transcriptions in a speech-to-text system
- etc.

In all cases we want to evaluate system in terms of ranking produced

Why learn to rank? – features

Each ranking task comes with many indicative features.

For **web search**, relevance indicators include:

- multiple retrieval functions (BM25, Embeddings-based, BERT,…)
- different document parts/views (titles, anchor-text, bookmarks, …)
- query-independent features (PageRank, HITS, spam scores, …)
- personalized information (user click history, …)
- context features (location, time, previous query in session, …)

Signals used by Search Engines

Search engines like Google combine **hundreds of signals** together

According to 2017 article (https://searchengineland.com/8-major-google-ranking-signals-2017-278450), Google's major ranking aspects were:

- incoming links:
  who links to the page? how relevant are those links (anchortext)?
- content:
  keywords, length, comprehensiveness
- technical quality:
  load speed, quality of mobile page
- past users:
  click through rate (CTR) from previous user searches

why learn to rank? – cont.

Given all these different signals,

- each of which can be indicative of relevance in the right context
- Question becomes: how best to combine the features?

- Impossible to combine them all by hand (too many of them)

Rank learning provides an **automated & coherent** method:

- for combining diverse signals into a single retrieval score
- while optimising a measure users care about, e.g. NDCG, ERR

formulating a learning problem

1. How can ranking be posed as a learning problem?
2. What are the examples and labels?
3. What is the loss function?

generating dataset: query + initial ranking

Query:

Tourism Amsterdam

1. Start with a query
2. Generate initial ranking using keyword-based ranker

	bm25
doc 1	108
doc 2	106
doc 3	92
doc 4	88
doc 5	43
doc 6	12
doc 7	4
doc 8	3
doc 9	2
doc10	1
doc11	1
doc12	1

generating dataset: truncate @ *k*

Query:

Tourism Amsterdam

	bm25
doc 1	108
doc 2	106
doc 3	92
doc 4	88
doc 5	43
doc 6	12
doc 7	4
doc 8	3

k

1. Start with a query
2. Generate initial ranking using keyword-based ranker
3. Truncate ranking as candidates for re-ranking

generating dataset: compute features

Query:

Tourism Amsterdam

1. Start with a query
2. Generate initial ranking using keyword-based ranker
3. Truncate ranking as candidates for re-ranking
4. Calculated feature values for each candidate

	bm25	bm25_title	anchortext	PageRank	...
doc 1	108	23	23	0.02	
doc 2	106	12	49	0.04	
doc 3	92	35	11	0.11	
doc 4	88	1	33	0.005	
doc 5	43	7	1	0.35	
doc 6	12	1	0	0.21	
doc 7	4	3	20	0.19	
doc 8	3	0	4	0.55	

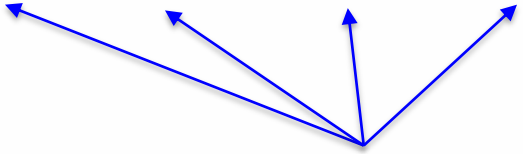
generating dataset: normalise features

Query:

Tourism Amsterdam

1. Start with a query
2. Generate initial ranking using keyword-based ranker
3. Truncate ranking as candidates for re-ranking
4. Calculated feature values for each candidate
5. Normalize each feature at the query level

	bm25	bm25_title	anchortext	PageRank	…
doc 1	1.00	0.66	0.47	0.03	
doc 2	0.98	0.34	1.00	0.06	
doc 3	0.85	1.00	0.22	0.19	
doc 4	0.81	0.03	0.67	0.00	
doc 5	0.38	0.20	0.02	0.63	
doc 6	0.09	0.03	0.00	0.38	
doc 7	0.01	0.09	0.41	0.34	
doc 8	0.00	0.00	0.08	1.00	



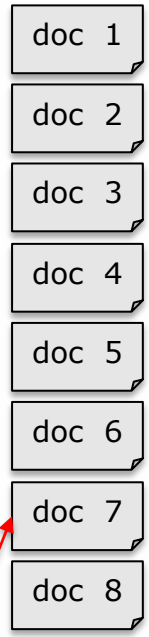
feature normalization

- usually perform min-max normalization at query-level for each feature
- necessary to make values **comparable across queries**

generating dataset: manual labelling

Query:

Tourism Amsterdam



1. Start with a query
2. Generate initial ranking using keyword-based ranker
3. Truncate ranking as candidates for re-ranking
4. Calculated feature values for each candidate
5. Normalize each feature at the query level
6. TRAINING: Provide ground-truth relevance labels for each query-document pair

bm25	bm25_title	anchortext	PageRank	...	RELEVANCE LABELS	
1.00	0.66	0.47	0.03		2	relevant
0.98	0.34	1.00	0.06		0	spam
0.85	1.00	0.22	0.19		1	not rel
0.81	0.03	0.67	0.00		3	highly rel
0.38	0.20	0.02	0.63		1	not rel
0.09	0.03	0.00	0.38		1	not rel
0.01	0.09	0.41	0.34		2	relevant
0.00	0.00	0.08	1.00		0	spam

query-document pair

feature vector

observed label

generating dataset: repeat for all queries

Query:

Tourism Amsterdam

	bm25	bm25_title	anchortext	PageRank	...	RELEVANCE LABELS	
doc 1	1.00	0.66	0.47	0.03		2	relevant
doc 2	0.98	0.34	1.00	0.06		0	spam
doc 3	0.85	1.00	0.22	0.19		1	not rel
doc 4	0.81	0.03	0.67	0.00		3	highly rel
doc 5	0.38	0.20	0.02	0.63		1	not rel
doc 6	0.09	0.03	0.00	0.38		1	not rel
doc 7	0.01	0.09	0.41	0.34		2	relevant
doc 8	0.00	0.00	0.08	1.00		0	spam

1. Start with a query
2. Generate initial ranking using keyword-based ranker
3. Truncate ranking as candidates for re-ranking
4. Calculated feature values for each candidate
5. Normalize each feature at the query level
6. TRAINING: Provide ground-truth relevance labels for each query-document pair

ice skating Rome

	bm25	bm25_title	anchortext	PageRank	...	RELEVANCE LABELS	
doc 1	1.00	0.36	1.00	0.39		2	relevant
doc 2	0.92	0.39	0.02	0.66		3	highly rel
doc 3	0.88	0.20	0.82	0.89		2	relevant

Gathering Relevance Judgments

Search engines employ people to annotate search results with relevance information!

- Google's detailed guidelines are for its Raters:
  https://www.hobo-web.co.uk/google-quality-rater-guidelines/

Other organisations (e.g. Wikipedia) can't afford to pay raters and try to collect judgments directly from users

- "Would this document have been relevant to query … ?"
  https://blog.wikimedia.org/2017/09/19/search-relevance-survey/

Usually don't train models directly from click data

- because causes a feedback loop

Evaluating Search Results









LOTS of different measures for evaluating retrieval results

- list some of the more popular ones

Traditional Measures:

- Precision at depth k

  What percentage of the top results are relevant?

  $$P@k = \#\{Relevant\ docs\ in\ top\ k\}/\ \mathrm{k}$$

- Recall at depth k

  What percentage of all the relevant documents available were found?

  $$R@k = \#\{Relevant\ docs\ in\ top\ k\}/\ \#\{Relevant\ docs\ in\ total\}$$

- F-measure at depth k

  Combining precision and recall, how well did we do?

  $$F_1@k = (\tfrac{1}{2}\ [(P@k)^{-1} + (R@k)^{-1}])^{-1}$$

Note: $P@k$ usually most important measure for retrieval

Evaluating Search Results (cont.)

Frequently used evaluation functions:

- MAP: Mean Average Position

$$\text{AveP} = \frac{\sum_{k=1}^{n} (P(k) \times \text{rel}(k))}{\text{number of relevant documents}}$$

  - terribly named measure!!
  - It's just average over queries of "Average Precision", which is average of P@k values at all rank positions containing relevant documents
  - Average Precision estimates area under precision-recall curve
  - MAP used to be most important measure

- NDCG@k: Normalized Discounted Cumulative Gain

$$\text{NDCG}(Q,k) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} Z_{kj} \sum_{m=1}^{k} \frac{2^{R(j,m)} - 1}{\log_2(1+m)}$$

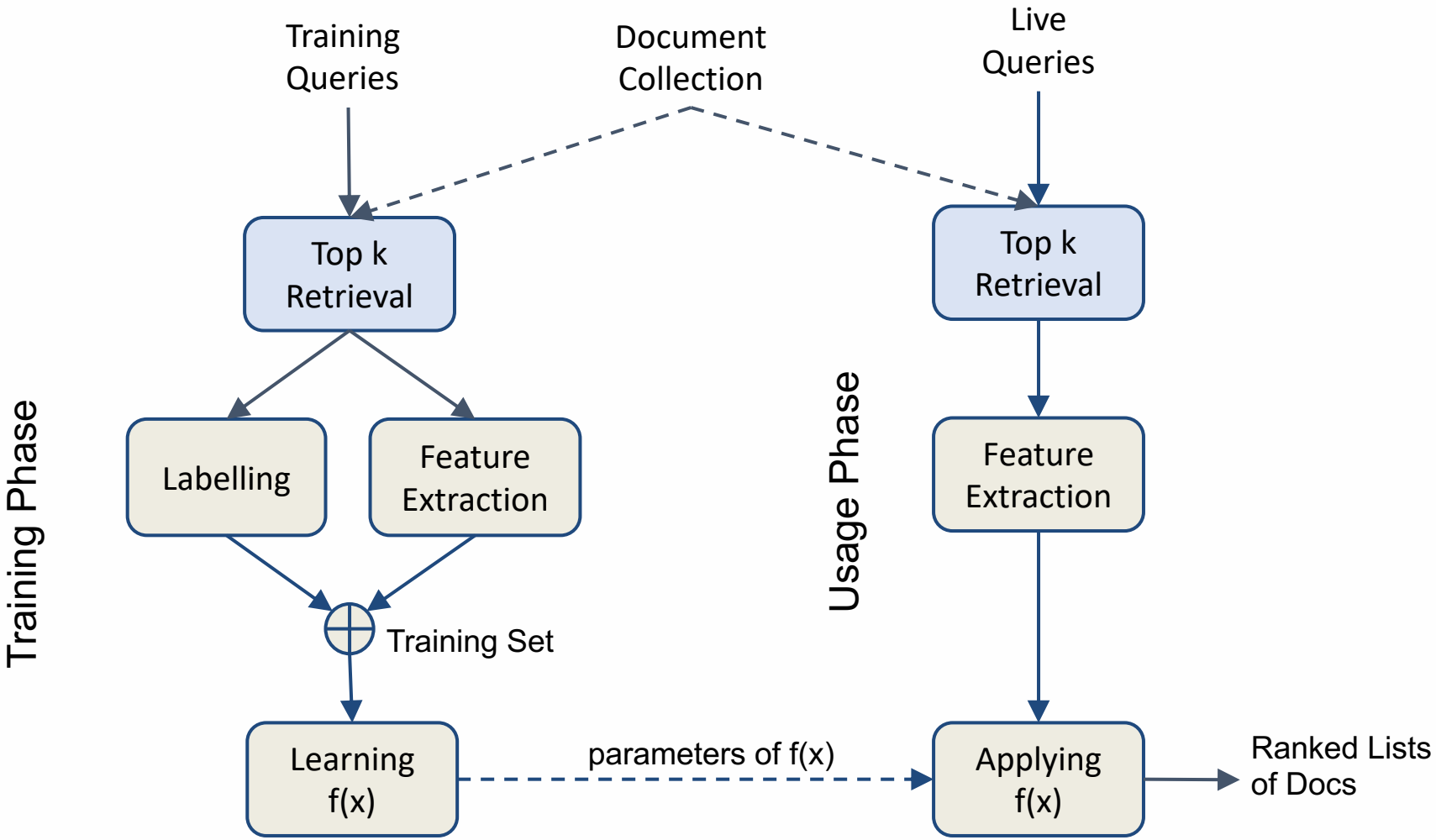
  - THE measure to care about by default
  - more faithful to user experience by discounting lower ranked docs
  - normalized at the query level

Evaluating Search Results (cont.)

Other common evaluation functions:

- ERR@k: Expected Reciprocal Rank
  - Especially useful for tasks with only 1 correct answer, e.g. known-item search, navigation style queries, …
- ER@k: Expected Rank
  - if know how frequently users bother to read second search result
  - used that probability to weight successful retrieval at that position.
  - so similar to NDCG but with empirical probabilities for discounting the importance of lower ranks….

two stage (re)ranking process



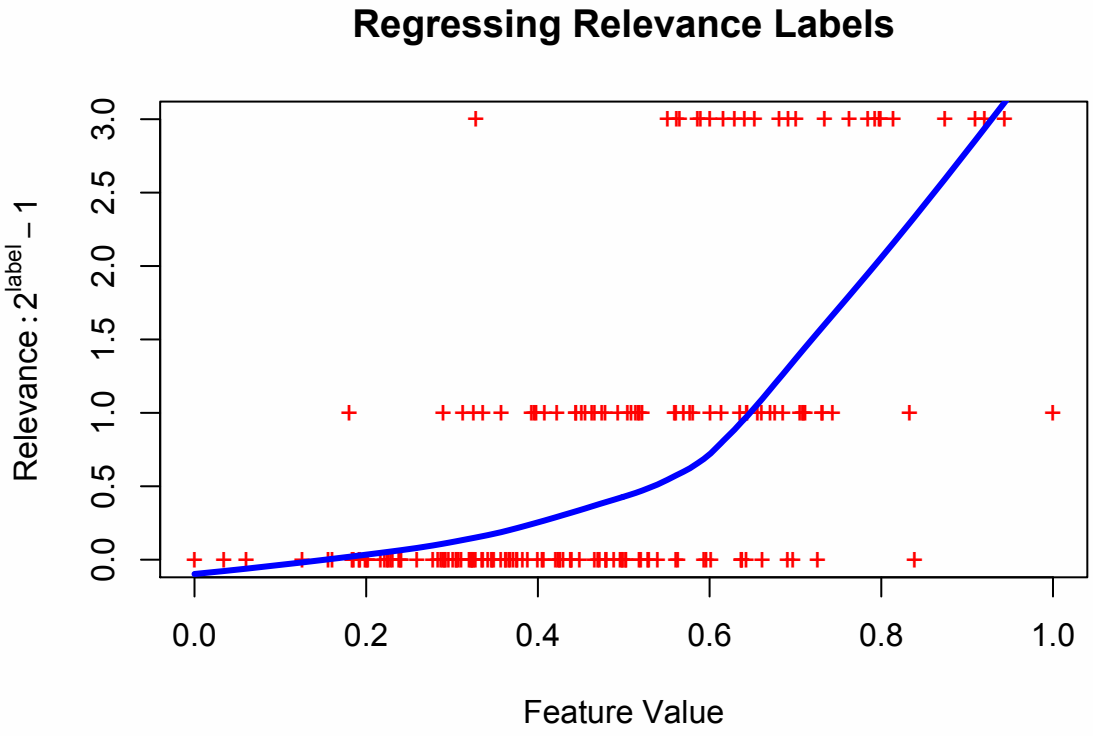
rank learning – treat as regression problem

Can treat rank learning as a simple regression problem:

- predict the relevance label based on feature values
- standard regression techniques can be applied

Caveat:

- optimising 'pointwise' objective is **suboptimal**, since it doesn't preference ordering **at top of list**
- ends up predicting well scores for less relevant documents



loss functions in learning to rank

During learning, loss function can be defined in 3 ways:

- Pointwise: (e.g. MSE)

x = feature vector

$f()$ = predicted score

y = relevance label

$$total\_loss = \sum_{i=1}^{m} \sum_{j=1}^{n_{q_i}} loss(f(x_i^j), y_i^j)$$

- Pairwise: (e.g. # incorrectly ordered pairs)

Predicted scores

Relevance labels

$$total\_loss = \sum_{i=1}^{m} \sum_{j=1}^{n_{q_i}} \sum_{k=j+1}^{n_{q_i}} loss(f(x_i^j), f(x_i^k), y_i^j, y_i^k)$$

- Listwise: (e.g. NDCG)

Predicted scores

Relevance labels

$$total\_loss = \sum_{i=1}^{m} loss(f(x_i^1), f(x_i^2), ..., f(x_i^{n_{q_i}}), y_i^1, y_i^2, ..., y_i^{n_{q_i}})$$

LambdaMART

- Listwise rank learner that makes use of the boosted regression trees
- Name comes from:

  Lambda (an approximation of loss gradient)

  + MART (Multiple Additive Regression Trees)
- Performs very well in practice
  - has become the default/baseline learner in most applications
- We'll now describe the algorithms in a bit more detail

Non-differentiable Loss Functions

For IR ranking learning problems

- we would like to set loss function to maximise evaluation measure of choice:
  $\mathcal{L}(y_1, ..., y_m, \hat{y}_1, ..., \hat{y}_m) = 1 - EvaluationMeasure(y_1, ..., y_m, \hat{y}_1, ..., \hat{y}_m)$

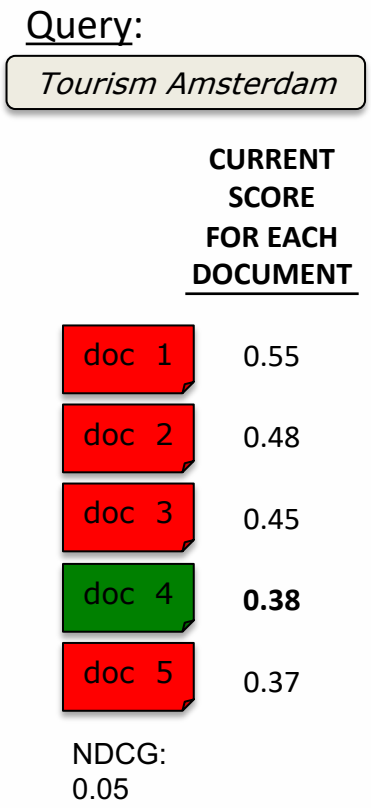
Problem:

- Ranking loss functions are non-differentiable (or flat) with respect to parameters of retrieval function

NDCG for example:

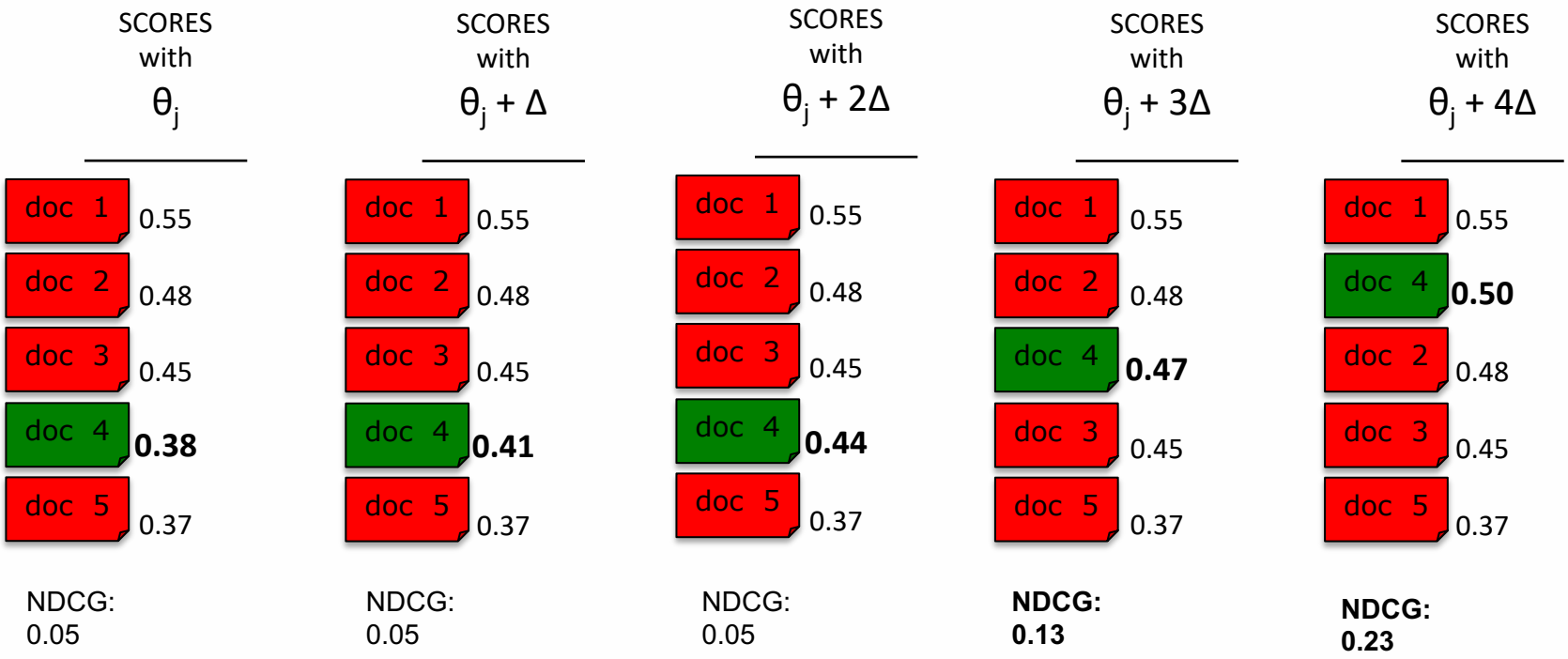
- is not a function of scores assigned to documents
  - but a function of sorted order induced by them
- so if parameter values change by only small amount
  - scores for documents will also only move a little
  - and ranking of the documents likely won't change

Understanding Ranking Loss



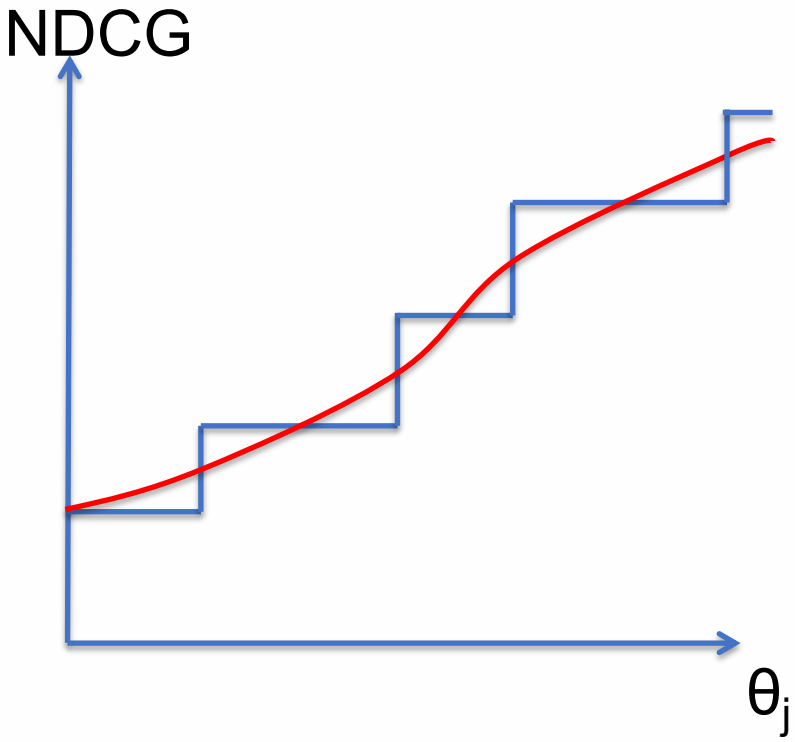
- Imagine have a ranking of 5 documents for a query
  - documents are ordered by their predicted scores
  - only fourth document is relevant to query
- Imagine modifying parameters of ranking function
  - such that the score for document 4 increases, while the scores for the others don't change
- E.g., document 4 could be longer than the others
  - and we increase a parameter that weights the document length
- Initially there is no change in order of documents
  - and thus no change in the evaluation measure!
  - eventually the score for document four will increase past the value for document 3

Understanding Ranking Loss (cont.)



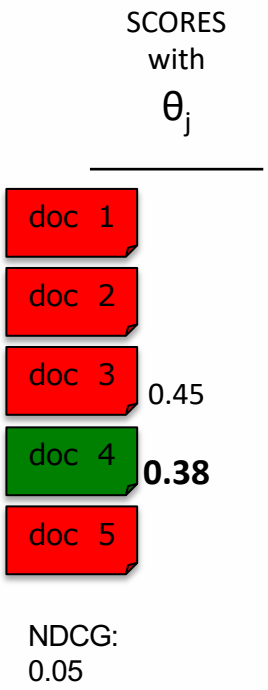
Note that we only see **change in the evaluation measure** (NDCG in this case) when the **ranking order changes**

Piecewise Flat Loss Function



- This effect results in a piecewise-flat loss function
  - no gradient information to let the learner know in which direction it should update the parameters
- LambdaMART effectively smoothes out this loss function

Understanding LambdaMART

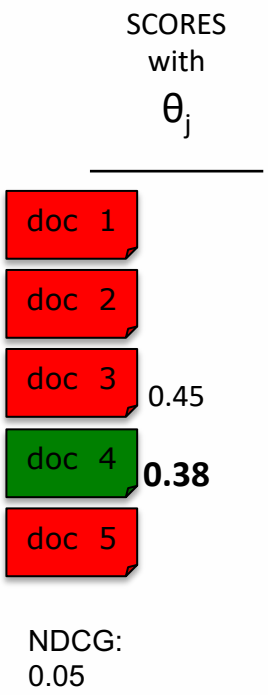


How does LambdaMART do this?

- Consider docs 3 and 4 in the ranking
  - The model's scores are only guesses as to the quality of the document
  - True quality of document is somewhere around that score, but could be higher or lower
- What is chance that the ordering should swap?
  - could use the logistic curve to model this probability
  - which depends on the difference between the scores

$$P(swap_{qij}) = \frac{1}{1 + \exp(\alpha(\hat{y}(x_{qi}) - \hat{y}(x_{qj})))}$$

Understanding LambdaMART



- If the two documents did swap
  - would effect Evaluation measure for ranking
  - increasing/decreasing it by

$$\Delta Z_{qij} = NDCG(swap_{qij}) - NDCG(\neg swap_{qij})$$

- So lambdaMART computes the expected change in the Evaluation Measure

$$\lambda_{qij} = \frac{-\alpha|\Delta Z_{qij}|}{1 + \exp(\alpha(f(x_{qi}) - f(x_{qj}))}$$

- and considers possibility of pairwise swapping of all documents in ranking

LambdaMART

- So we have the pairwise gradient

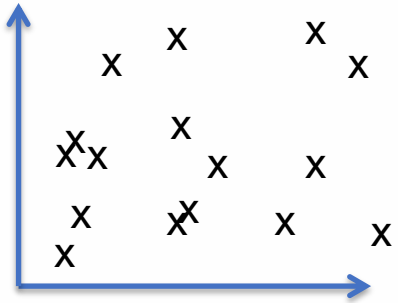
$$\lambda_{qij} = \frac{-\alpha|\Delta Z_{qij}|}{1 + \exp(\alpha(f(x_{qi}) - f(x_{qj}))}$$

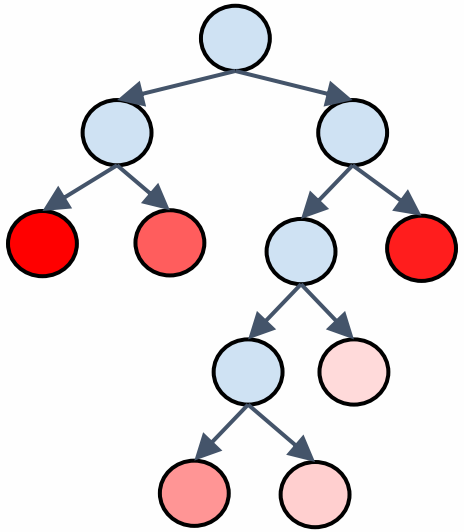
- Positive and negative affects of all pairwise expectations can be summed to produce the final (pointwise) gradient

$$\lambda_{qi} = \sum_j \lambda_{qij} - \sum_j \lambda_{qji}$$

- And a new regression tree is fit to these values and added to the ensemble …

$$score(q, x_{qi}) = \lambda_{qi}$$





More Resources for LambdaMART

- Some other resources describing the LambdaMART algorithm:
  - Overview paper (bit of complicated read) on LambdaMART by inventor of algorithm:
    - https://www.microsoft.com/en-us/research/publication/from-ranknet-to-lambdarank-to-lambdamart-an-overview/
  - Short blog post that gives some nice intuition:
    - https://medium.com/@nikhilbd/intuitive-explanation-of-learning-to-rank-and-ranknet-lambdarank-and-lambdamart-fe1e17fac418
  - Slides explaining LambdaMART:
    - https://staff.fnwi.uva.nl/e.kanoulas/wp-content/uploads/Lecture-8-1-LambdaMart-Demystified.pdf
  - Blog post on Visualising LambdaMART:
    - https://wellecks.wordpress.com/2015/02/21/peering-into-the-black-box-visualizing-lambdamart/

Conclusions

Conclusions

TODO