Information Retrieval AS 2015: summary

Taivo Pungas

February 8, 2016

1 Near-duplicate detection

Material from lectures: **2**

We want to detect which documents in a database are almost exact duplicates of each other.

Challenges: L2.5-6

1. Use a robust similarity measure w.r.t. small variations: whitespace, punctuation, misspeelling, capitalisation, small text insertions/deletions.
2. Avoid quadratic blow-up (all pairwise comparisons).

Solution: summarise documents into sketches that can be efficiently clustered into near-duplicates. L2.6

Possible sketch: character n-grams L2.6

- Pros: no tokenisation needed, language agnostic
- Cons: not space-efficient, hard to tune trade-offs, no inherent semantics (compared to words)

Shingles (word q-grams) work better. Each document is represented by the set of q-length token sequences that occur in it.

Jaccard coefficient as similarity measure: L2.7

$$\mathrm{sim}(A, B) := J(A, B) = \frac{|A \cap B|}{|A \cup B|} \in [0, 1]$$

1.1 Similarity hashing

If we choose the right hash function h, then $J(S_A, S_B) = Pr[h(S_A) = h(S_B)]$. If we sample enough different h-s, we get an accurate estimate of $J(S_A, S_B)$. Special case of locality-sensitive hashing.

MinHash: early approach, hard to apply in practice, heuristic approach used. L2.7 SimHash: better approach, combines hashed shingles into a single bitstring by majority voting. See L2.8.

Use sorted tables of permuted fingerprints to do quick near-duplicate detection given a query doc.

2 Scoring search results

Material from lectures: **3-5**

2.1 Preprocessing documents and queries

We can preprocess documents and queries by:

- Stemming aka lemmatisation: reducing words to their stems. E.g. Porter stemmer. L2.4
- Tokenising: e.g. handling numbers/dates/names/phrases as special cases. L2.4
- Removing stop words (common terms like "the", "a", etc). L2.4

How to order the documents our search engine retrieved such that the most relevant ones come first? Use a score function that assigns a relevancy score to each document, and produces a result list by returning documents with the highest scores.

2.2 Evaluating scoring functions: precision and recall

How to get ground truth data about relevance? We could get opinions from humans, but there are many challenges L3.1. Still widely used though. We can also infer relevance from user click behavior.

When we have ground truth data, we can use precision and recall metrics: L3.1

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{\#\text{ relevant items retrieved}}{\#\text{ items retrieved}} \tag{1}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\#\text{ relevant items retrieved}}{\#\text{ relevant items in collection}} \tag{2}$$

Precision/recall curve: for each i, take top i results, calculate precision and recall and plot result. P/R curves also generalise to ranked lists, not just sets, which makes them better for assessing search results. L3.2-3

In web search, we want high precision since user looks at mainly top 2-3 results. Lower recall is more tolerable. There are also use cases where we might want to have very high or perfect recall. L3.2

Evaluating other things than precision/recall We can A/B test two systems: users tell us which result page is better. This allows us to compare whole result *pages*, identify failures and also combine it with relevance assessments. Possible to do importance sampling (ask users to provide assessments that are especially important to us, e.g. ones we're most unsure about). L3.3

2.3 Naive scoring

$\text{Score} = \#\text{ common terms between query and document}$, with tie-breaking. L2.8-9

2.4 tf-idf scoring

The tf-idf score takes into account two things: L4.1-3

- How much the query term occurs in the given document (term frequency).
- How common the query term is in the entire collection (document frequency).

Empirically, raw term frequencies give bad weights. For this reason, we apply a shift and logarithm: L4.1

$$\text{log-tf}(w; d) = \log(1 + \text{tf}(w; d))$$

Query terms that are found in almost all documents are less informative, so we want to decrease the weights of more common query terms. Let's take the inverse of document frequencies and logarithm it (common way to do it) L4.2. Assume we have n documents.

$$\text{df}(w) = \#\ \{\text{documents that contain } w\} \leq n$$
$$\text{idf}(w) = \log \frac{n}{\text{df}(w)}$$

idf can also be justified information-theoretically L4.2. Instead of df, we could use collection frequency, but it has been shown to perform worse L4.2.

The tf-idf weights are given as a product:

$$\text{tf-idf}(w; d) = \text{log-tf}(w; d) \cdot \text{idf}(w; d) = \log(1 + \text{tf}(w; d)) \cdot \log \frac{n}{\text{df}(w)}$$

The final score of document d is the sum of weights in the query q: L4.3

$$\text{score}(d, q) = \sum_{w \in q} \text{tf-idf}(w; d)$$

We can also treat both documents and queries as vectors in a high-dimensional space of tf-idf weights where each dimension is a term (the vectors are sparse and dimensionality is very high.) Then we can rank documents according to their proximity to the query: L4.3

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

In practice we use non-symmetric scoring functions.

2.5 Language models for scoring

Let's assume the user imagines a relevant document and picks random terms out of it to form a query. Then we want to estimate the distribution of query terms given a document, $P(w|d)$. If we have estimated this, the query probability for a document is

$$P(q|d) = \prod_{w \in q} P(w|d)$$

which becomes the scoring function – we show the document that was most likely to generate this query. L5.1

Pros: simple, easy to estimate $P(w|d)$ based on document collection we have. Cons: too strong assumptions, relevance too strictly tied to probability. L5.1

How to estimate $P(q|d)$ Trivial approach: using term frequency counts L5.1. This overfits, though (especially for words that occur very rarely) so we can use smooth using collection frequencies (Jelinek-Mercer smoothing, L5.1):

$$P(w|d) = (1 - \lambda_d)\hat{P}(w|d) + \lambda_d \hat{P}(w)$$

where λ_d can depend on e.g. the length of document d. If we differentiate between terms that occur in the document and those that don't, we have a more general approach. L5.2

Smoothing is not magic though, it just helps with rare terms. Instead, we can use topic models. L5.2

2.6 Topic models for scoring

Each document is assumed to be generated as a mixture of topics (using a latent variable for topic assignment). This allows us to express $P(w|d)$ by summing out the latent variable: L5.2

$$P(w|d) = \sum_t P(w|t)P(t|d)$$

Two extremes of topic modelling are: a) assuming only one topic (equivalent to a collection model) and b) assuming same amount of topics as documents (equivalent to MLE model).

If we don't know $P(w|t)$, we can estimate it using our collection: iteratively updating our estimate using a multiplicative update rule L5.4.

To estimate $P(t|d)$, we can also do multiplicative updates. This converges quickly and can be proven from the Expectation Maximisation algorithm. L5.3.

We can get a generative model (i.e. find a probability distribution over documents) of documents using Latent Dirichlet allocation: L5.5

$$P(d|params) = \int_\pi \{\text{prior on such a } \pi\} \cdot \{\text{likelihood of d given } \pi\}$$

3 Text categorisation

Material from lectures: **6**

Applications: L6.2

- Sort documents into categories.
- Route messages to appropriate employee.
- Language identification.
- Sentiment detection.
- Relevant/irrelevant classification in search results.

Can categorise based on topic, genre, function, author, style, dichotomy (spam/ham) etc.

Problems of extracting rules from human experts: hard to verbalise, low coverage, not very accurate. L6.2

Optimal Bayes classification maximises the posterior probability $P(class|document)$: L6.3

$$c^* = \arg\max_c P(c|d) = \arg\max_c P(d|c)P(c)$$

Naive Bayes classification approximates

$$P(d|c) = \prod_{w \in d} P(w|c)^{\mathrm{tf}(w;d)}$$

We don't know $P(c)$ and $P(w|c)$, but we can estimate them from our collection L6.3, which yields the following classification rule (if we logarithm the posterior first):

$$c^* = \arg\max_c [\log \hat{P}(c) + \sum_{w \in d} \mathrm{tf}(w;d) \log \hat{P}(w|c)]$$

This approach runs into a problem: many $\hat{P}(w|c)$ are equal to 0 $\implies$ documents containing unseen words (for a class) will never be assigned to this class. Solution: Laplace smoothing ($\alpha > 0$): L6.4

$$\hat{P}(w|c) := \frac{\sum_{d \in c} tf(w;d) + \alpha}{\sum_{d \in c} len(d) + \alpha \#\{w\}}$$

We can evaluate the resulting classifier using hold-out data (cross-validation).

The Naive Bayes classification rule is linear, so we can generalise to logistic regression and learn the optimal parameters directly – the features in our training data are term frequencies L6.5. In logistic regression, we can run online stochastic gradient descent so it is possible to learn in a streaming fashion L6.5. It can be generalised to multiclass classification using the softmax function L6.6.

SVMs are another option for classification; they can also be trained online using the PEGASOS algorithm with a stochastic update rule L6.7.

4 Machine learning for ranking

Material from lectures: **7**

Goal: learn a ranking function, given training data.

Types of training data: L7.1

- Pointwise: relevance of individual documents.
- Pairwise: document A is preferred to document B.
- List data: ideal ranked list (NOT covered in lecture).

Since relevance is always relative to a query q, we need to extract features from document-query pairs (d, q). We can get these features from several sources L7.1.

For pointwise data we can use linear ordinal regression, which is a generalisation of binary classification: it lets us assign each datapoint into one of R buckets called *regression levels*. The real line is partitioned into R intervals, each of which is a regression level. We can define a generalised hinge (margin) loss, calculate the subgradient and then run online stochastic subgradient descent. L7.1-2

For pairwise data we can similarly define a simple regression function, define a pairwise loss and run online stochastic subgradient descent. This method is especially well-suited if we only have user click data and no explicit relevance ratings. L7.3

5 Document indexing

5.1 Indexes

Material from lectures: **8**

For real-time search systems, we want to keep an efficient look-up structure for retrieving and scoring documents. For this, we invest offline computation and storage to make queries faster. Main challenges: distributing over machines, scalable construction, support for different queries, compression. **L8**.1

Types of indexes A *forward index* is a mapping $(\text{document ID} \rightarrow \text{list of terms})$ **L8**.1. An *inverted index* is the reverse of a forward index: it is a mapping $(\text{term} \rightarrow \text{list of document IDs})$ **L8**.2. A *frequency index* is an inverted index with term counts **L8**.3. A *positional index* is an inverted index with term positions (and, implicitly, term counts) **L8**.5.

It is also possible to index phrases (*multi-term indexing*); this is faster than positional indexing **L8**.5 and done by identifying phrases during tokenisation.

List intersection For querying, we need to quickly intersect lists of document ID-s (one for each query term) that we got from the inverted index. If the lists are sorted, we can exploit this to create a more efficient list intersection algorithm. For intersecting multiple lists, starting with the shortest lists is more efficient. **L8**.3-4

5.2 Scaling up indexing

Material from lectures: **9**

Our collection (the Web) is too large to fit an index in the memory of a single machine. There are three possible solutions: BSBI, SPIMI and MapReduce.

Blocked Sort-Based Indexing (BSBI) processes postings in blocks and then merges them **L9**.1. *Single-Pass in-Memory Indexing* (SPIMI) partitions the collection to many machine and constructs local indexes for each partition, which can then be merged into a global index (at query time or before) **L9**.1.

MapReduce **L9**.2

- Mapper: emits postings (term, docID, 1).
- Combiner: combines duplicate postings (sums counts).
- Reducer: groups tuples with the same key (term) and returns list of documents containing that term.

5.3 Serving indexes

Material from lectures: **10**

When we have an index, how do we store it and process queries? We need to split the index into pieces aka shards. In *document sharding*, each machine gets a range of document IDs; in *term sharding*, each machine gets a range of terms L10.1. Both have pros and cons L10.2; document sharding is standard.

A common approach is to assign smaller IDs to more important documents and partition index by document. Index shards can be replicated to increase query capacity; a cache layer can be added on top of this. L10.2 If we have enough shards (or replicas?), we can fit the whole index into memory L10.3.

5.4 Real-time scoring

Material from lectures: **10**

How to calculate score of documents once we have an inverted index? In several phases: a) get candidate documents, b) preliminary scoring, c) full scoring, and more. This can be done in a distributed manner; then we need to combine the result lists. L10.3

Since we don't want to retrieve documents with small scores, we can discard many documents. There are also several heuristics how to reduce the size of the result list, like taking the union of pairwise 2-word subqueries L10.3, or ordering posting lists by PageRank (or something else) and aborting search after a partial scan of the list L10.4.

5.5 Dictionaries

Material from lectures: **10**

Heap's law tells us the vocabulary size (number of distinct tokens m) is $O(\sqrt{n})$ in the number of tokens n. Specifically, $m(n) = kn^\beta$ where $\beta \approx 0.5$ and $30 \leq k \leq 100$ L10.4. Since the dictionary can get quite large at web-scale and the uncompressed fixed-width array representation is wasteful, we might need to compress it. There are several ways to compress a dictionary: string representation, block-string representation or hash tables L10.5.

5.6 Index compression

Material from lectures: **10**

Indexes take a lot of space (disk or memory). Zipf's law tells us that collection frequency of terms follows a power-law distribution: there are few very frequent terms and many rare terms. This means that for frequently-occurring terms, gaps between consecutive docID-s are much smaller than the docIDs themselves L10.6. Shannon's entropy lower-bounds our average code length L10.8.

Variable Length Encoding (VLC) lets us take advantage of the fact that small gaps are very common. VLC uses 7 bits to encode a number and 1 bit as the continuation bit which tells us whether the next block (8 bits) contains a new number or continues the current number. It is very simple and efficient, but limited to 8-bit (or multiple) codewords. L10.6

Gamma code gives us bit-level control over codeword length. We represent each number by its length (in *unary*) concatenated with its offset (in binary) L10.6. Examples in L10.7. Downside: decoding can be slow (higher query time) and the process is complex.

Golomb code makes some simple assumptions and yields a very efficient Huffman-style code given these assumptions. It uses a divisor b and encodes a number x as integer quotient q and remainder r such that $x = qb + r$. L10.8

An extension, the Local Bernoulli model, uses a different b for each term: small b for frequent terms (small gaps between docID-s) and large b for infrequent terms (large gaps). It is better than global Bernoulli in practice. L10.9

Some further optimisations can be made: block-based variable-length encoding, Varint encoding, group Varint encoding. L10.9

6 Link analysis

Material from lectures: **11**

We'd like to compute relevance of documents without queries. For this, we can exploit the hyperlink graph of the Web. Thus we make two assumptions: a) hyperlink is a quality signal (endorsement) and b) anchor text describes the target page. This may be suscepible to manipulation though (link bombs). **L11**.1

PageRank We can think of the web as a directed graph: nodes are pages, links are edges. In this case we can do citation analysis, e.g. calculate the [weighted] citation frequency for each node **L11**.2. This is the idea of *PageRank*.

To use the in-degree as a quality score, we can calculate PageRank using random web surfing. By defining transition probabilities at each node we get a Markov Chain; the stationary distribution of this MC encodes the importance (PageRank) of each page. **L11**.3

Instead of sampling the MC step-by-step, we can use the *power method*: simulate all one-step transitions simultaneously by multiplying our estimate by the adjacency matrix T in each step. This means we just need to find the eigenvector of T. **L11**.3

To personalise PageRank, we can calculate a PageRank for each topic and model users as weighted combinations of topics **L11**.4.

The hub-and-authority model assumes there is two types of relevance: a) hub – a good lists of links to information needed, and b) authority – a direct answer to the information need. We can calculate hub-and-authority scores on the top n search results (the root set) by finding all in/out-neighbors of the root set (base set) and iteratively updating the hub scores and authority scores **L11**.5-6. The updates are done based on the adjacency matrix and we can apply the power method here as well **L11**.7.

Comparing PageRank and hub-authority (HA) model: **L11**.7

- PageRank can be precomputed, HA needs expensive query-time computation.
- They formalise the eigenvalue problem differently.
- They use a different set of pages to apply the formalisation to.

7 Recommender systems

Material from lectures: **12**

Recommender systems are designed to automatically give personalised recommendations to users based on other users' choices (collaborative filtering) or on attributes of items previously chosen by the same user (content-based filtering). **L12**.1-2

The data we can use can be explicit (ratings, reviews, questionnaires) or implicit (page visits, purchase history, clicking patterns). We can also distinguish between short-term tasks a customer is trying to complete and long-term interests. **L12**.1-2

Collaborative filtering has some advantages over content-based filtering: we don't need to be able to characterise the items; we can filter based on quality and taste (not just content) **L12**.2.

Matrix completion is the main task in collaborative filtering. Using the Frobenius norm (element-wise squared error summed over the matrix) and assuming the matrix has low rank **L12**.2, we can a) impute and run SVD or b) use a Latent Vector Model (LVM) **L12**.3. The LVM assigns a vector to each item (L) and each user (R), predicts ratings as a scalar product of these vectors, and is trained using alternating least squares (update L holding R fixed, then update R holding L fixed). The training of LVMs is easy to parallelise but only finds local optimum. **L12**.3

8 Entities and semantic search

Material from lectures: **13**

An entity is *a thing with a name*. Entities have IDs, names, types, attributes, and relationships **L13**.1. Challenge with entity name mining: name disambiguation, multilingual names **L13**.2. Entities can be linked to other entities through different relationships.

In any text, we can find and link entities. We can detect which words represent other entities by doing statistics: keyphraseness and commonness **L13**.3. To disambiguate, we can compare the local context in our text, and the Wikipedia article on each of the candidate entities.

We can use relatedness – defined using inbound links to entities – for disambiguating entitiy names in articles. **L13**.4

Having a good database of entities allows us to respond to search queries by understanding user intent. **L13**.5