Information Retrieval AS 2015: summary

Taivo Pungas

February 7, 2016

1 Near-duplicate detection

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

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 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: $\boxed{\text{L4.1-3}}$

- 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: $\boxed{\text{L4.1}}$

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

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) $\boxed{\text{L4.2}}$. 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 $\boxed{\text{L4.2}}$. Instead of df, we could use collection frequency, but it has been shown to perform worse $\boxed{\text{L4.2}}$.

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: $\boxed{\text{L4.3}}$

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

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: $\boxed{\text{L4.3}}$

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

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

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

1.

•