Information Retrieval
Extending Vector Space Model

Vector Space Model for IR can be extended/augmented in a number of ways:

- Feedback processing.
- Thesaurus for matching synonyms/similar words.
- Inverted Indexes for improving retrieval speed.
- Postings files for proximity queries.

Feedback Processing

Relevance feedback processing is a collection of IR techniques that use *relevance judgements obtained from humans*[1] to refine (and, in theory, **improve**) the results of a retrieval procedure.

Process. Let q be a user query to a document collection D. Suppose the query returns the set D_q of documents.

A human analyst (user of the IR system) then examines some of the documents in the set D_q and identifies two sets D_q^r: a set of *relevant documents* and D_q^{irr}: the set of *irrelevant documents*. Note that, we expect that

$$D_q - (D_q^r \cup D_q^{irr}) \neq \emptyset.$$

Idea. Change the representation of the query q to retrieve more documents like those in D_q^r, exclude documents that are in D_q^{irr}, and not retreive documents similar to them.

[1] Usually, those who originated the IR query.

Rocchio relevance feedback processing method. The query vector q is replaced with a new vector q_e which:

- Emphasizes the keywords found in documents from D_q^r.
- De-emphasizes the keywords found in documents from D_q^{irr}.

$$q_e = \alpha \cdot q + \frac{\beta}{|D_q^r|} \sum_{d_r \in D_q^r} d_r - \frac{\gamma}{|D_q^{irr}|} \sum_{d_i \in D_q^{irr}} d_i.$$

Here, α, β and γ, often taken so that $\alpha + \beta + \gamma = 1$, represent respectively, *the importance of the original query*, *the importance of the positive information* and *the importance of the negative information.*

Notes. Rocchio feedback processing introduces the potential of ***negative keyword weights***. A negative keyword weight in a query vector means that *the lack of that keyword in a document is important w.r.t. the relevance judgement*.

Additionally, if $\alpha + \beta + \gamma > 1$, the absolute values of keyword weights will grow (especially after a few iterations of the feedback method).

Variations. A number of variations on Rocchio's method.

- No negative feedback: $(\gamma = 0)$

$$q_e = \alpha \cdot q + \frac{\beta}{|D_q^r|} \sum_{d_r \in D_q^r} d_r.$$

- Diminished negative feedback. Only use one vector from D_q^{irr}:

$$q_e = \alpha \cdot q + \frac{\beta}{|D_q^r|} \sum_{d_r \in D_q^r} d_r - \gamma \cdot d_{irr}^{max},$$

  where $d_{irr}^{max} \in D_q^{irr}$ is the highest ranked irrelevant document.

Blind Relevance Feedback . Otherwise known as **pseudo relevance feedback**. Let IR system retreive the set D_q of documents given query q. Assume that the top $k << |D_q|$ documents are **relevant** and perform Rocchio's feedback (w/o the negative information) transformation of q.

This is similar to **boosting**.

Use of Thesaurus

All methods discussed thus far will retrieve a document if it contains **at least one keyword (stem)** specified in the query.

Thesauri help alleviate this issue.

Simple Thesaurus. A **simple thesaurus** is a collection of triples

$$(t_i, t_j, \alpha),$$

where $t_i, t_j \in V$ are two terms from the vocabulary and $\alpha \in (0, 1]$ is the *degree of similarity*.

If $\alpha = 1$, t_i and t_j are **exact synonyms**. E.g. *("person", "human", 1.00)* means that words *"person"* and *"human"* should be treated as full synonyms.

If $\alpha < 1$, it means that t_i and t_j are similar, but their similarity does not rise to the level of complete synonymity. E.g., we can have *("car", "Toyota", 0.5)*, because we know that a *"Toyota"* is (typically) a car, but not every *"car"* is a Toyota.

Computing similarity. In the presence of a simple thesaurus, we need to compute similarity between a document and a query in a different way. Let $T = \{(t_i, t_k, \alpha_{ik}\}$ be a simple thesaurus.

$$sim(d_j, q) = \frac{\sum_{i=1}^{M} d_{ij} \cdot q_i + \sum_{(t_i, t_k, \alpha_{ik}) \in T} \alpha_{ik} \cdot d_{ij} \cdot q_k}{\sqrt{\sum_{i=1}^{M} d_{ij}^2 \cdot \sum_{i=1}^{M} q_i^2}}.$$

Note. We can treat a simple thesaurus as both *symmetric* and *assymetric*. If a simple thesaurus is **symmetric**, then $(t_i, t_k, \alpha) \in T$ implies that $(t_k, t_i, \alpha) \in T$. If a simple thesaurus is **assymetric**, then $t_i, t_k, \alpha) \in T$ does not imply $(t_k, t_i, \alpha) \in T$. In this case it is possible that $(t_k, t_i, \alpha') \in T$ for some $\alpha' \neq \alpha$, or that there is no entry of the form $(t_k, t_i, .)$ in T at all.

In all cases, the formula above will work.

Inverted Indexes and Postings Files

Without special preparations, each time a query q is given to an IR system, the system must compute and sort all $sim(d_1, q), sim(d_2, q), \ldots, sim(d_n, q)$. When n is very large, this is a costly operation.

Inverted Index is a data structure that allows for more efficient query processing.

A collection of document vectors $D = \{d_1, \ldots, d_n\}$ can be thought of as a mapping from document Ids $(d_1, \ldots, d_n)$ to term ids $t_1, \ldots, t_m$.

An **inverted index** is a mapping from **terms** to **documents** that contain them.

Simple Inverted Index is a list $\{\langle t_i, (d_1^i, \ldots, d_{k_i}^i)\rangle\}$, where $t_i \in V$ is a vocabulary term and $d_1^i, \ldots, d_{k_i}^i$ are **all** documents in D that contain t_i.

Example. Consider the following three documents:

d_1 *When I say stop, continue.*

d_2 *When I say stop, stop and turn around.*

d_3 *Around the bend, the river continued.*

Assume for a moment that stopword removal removes *"the"* and *"and"* and that *"continued"* stems to *"continue"*. Then, the **simple inverted index** for this document collection will be:

`when`	d_1, d_2
`I`	d_1, d_2
`say`	d_1, d_2
`stop`	d_1, d_2
`continue`	d_1, d_3
`turn`	d_2
`around`	d_2, d_3
`bend`	d_3
`river`	d_3

Search using Inverted Index. Let D be a document collection, V be its vocabulary, and I be its inverted index. Let $I(t_i)$ denote the list of documents that contain t_i. Given a query q, its evaluation can proceed as follows:

- **Step 1: Listings.** For each query term t_i present in q, retrieve $I(t_i)$.
- **Step 2: Merge.** Compute the intersection of all retrieved $I(t_i)$s. (If necessary, compute the union of $I(t_i)$s and sort it according to the number of matching terms in each document).
- **Step 3: Rank.** For each document d_j from the list computed on Step 2 compute $sim(d_j, q)$. Sort all documents in the descending order of the similarity.

Inverted Indexes with Postings Files

An **inverted index** can be adapted to help deal with proximity queries.

Postings. A **posting** is a triple (t_i, d_j, k), which specifies that term t_i occurs in document d_j in position k. **Position** is usually defined as the word position (order) in the document after stopword removal.

Inverted index with postings file: an **inverted index** where for each indexed document we specify all locations of the term in it. More formally, an **inverted index with postings file** is a collection of tuples of the form

$$\langle t_i, \langle d_1^i, (k_{11}, \ldots, k_{1s_1})\rangle, \ldots, \langle d_{l_i}^i, (k_{l_i1}, \ldots k_{l_is_{l_i}})\rangle\rangle.$$

Here, $d_1^i, \ldots, d_{l_i}^i$ are **all** documents from D which contain term t_i, and k_{rt} are all the locations in which the terms occur in their respective document.

Example. The **inverted index with postings file** for the document collection

d_1 *When I say stop, continue.*

d_2 *When I say stop, stop and turn around.*

d_3 *Around the bend, the river continued.*

will be:

`when`	$(d_1, 1), (d_2, 1)$
`I`	$(d_1, 2), (d_2, 2)$
`say`	$(d_1, 3), (d_2, 3)$
`stop`	$(d_1, 4), (d_2, 4, 5)$
`continue`	$(d_1, 5), (d_3, 4)$
`turn`	$(d_2, 6)$
`around`	$(d_2 7), (d_3, 1)$
`bend`	$(d_3, 2)$
`river`	$(d_3, 3)$

Proximity queries. **Inverted indexes with postings files** can be used to answer exact phrase queries and proximity queries.

Let D be a document collection, V be its vocabulary, and I be its inverted index with postings. Let $I(t_i)$ denote the list of documents that contain t_i and $I(t_i, d_j)$ denote the list of postings for t_i and d_j. Given a query q that represents the exact phrase to be match (or a collection of keywords that need to be found in close proximity), the search will proceed as follows.

- **Step 1: Listings.** For each query term t_i present in q, retrieve $I(t_i)$.
- **Step 2: Merge.** Compute the intersection of all retrieved $I(t_i)$s.
- **Step 3: Filter.** For each document d_j in the list computed on Step 2, establish the proximity of keywords. If the proximity test fails, remove the document from the list.
- **Step 4: Rank.** For each document d_j from the list computed on Step 3 compute $sim(d_j, q)$. Sort all documents in the descending order of the similarity.