Wildcard Queries

Wildcards simply means substitute anything to `*`: "e*a" is anything containing "e" and "a". It is used when the user is uncertain of the spelling of a word (e.g. American and English spelling), or we want to catch all variants of a term.

Recall the process of stemming: we can have a superposition between the two features (they can interact).

The simplest case is **trailing wildcard**: `term*` there is only a wildcard symbol and it is at the end of the word (search string). Recalling the structures (trees, B-trees) we saw, all terms are inside a collection of subtrees, so a search tree on the dictionary is a convenient way of handling trailing wildcard queries.

Once we find all the terms satisfying the wildcard we perform the union of all them to answer the query.

The **leading wildcard**: `*term` there is only one wildcard and it is at the beginning of the word. In order to answer, we can build a new data structures where everything is ordered but in reverse, then finding which terms satisfy the trailing wildcard of the reverse.

Now we need to solve the problem of one wildcard, in any position (**general wildcard queries**). We want to drive this problem to the trailing or leading wildcard, that we are able to solve.

The idea behind the techniques to solve this problem is to express the given wildcard query as a boolean query on a specially constructed index, such that the answer is a superset of the set of vocabulary terms matching the wildcard query. The we check every term in the answer against the wildcard query, discarding the ones that do not match it. Having the vocabulary terms matching the wildcard query, we can resort to the standard inverted index.

Permuterm index

We can add to the inverted index all the rotations of the word. Meaning that we add a special *end of the word* symbol and we consider the rotations. For each of these rotations, we have that they point to the same posting list (they are the same word, just begin to read after the end symbol). If we have a word of n character we have $n + 1$ rotations, because of the ending character.

We need to add the end of the word symbol at the end of the query: e.g. `C*T` $\rightarrow$ `C*T$`, and then we rotate the word so to have the wildcard at the end: `C*T$` $\rightarrow$ `T$C*`.

In this way we have more terms but the same number of postings as before. We are able to perform queries with one wildcard, in any position.

We need now to solve the problem of multiple wildcards, we will use the permuterm index plus a post-processing step. We first put the end of the word

symbol at the end of the word, `*A*T` → `*A*T$`, then we consider the more general query where everything between the first and the last wildcard is folded inside a single wildcard, `*T$` → `T$*` i.e. at the end we rotate to have a trailing wildcard query. Then we filter the answer to answer the original query: it is expensive, but most of the queries use only a single wildcard.

There is an interplay between the algorithm that we use and the data structure employed. The drawback is that the permuterm index requires a huge amount of space.

K-gram indexes

The idea is to have two indexes, one for the wildcard queries, one for the normal queries. The **k-gram** is a sequence of k characters, we don't index the terms, but the k-grams. Each k-gram points to all the terms that contains that k-gram. Now we have two levels of indexing: the k-grams and an index of the terms that points to the postings. So the k-grams allow us to find the terms that allow us to find the postings and finally the documents.

So in order to answer the queries, we need to add `$` as beginning and end of word symbol, we extract the k-grams for a given k, we search for the k-grams and finally we intersect the results.

A problem is that we may also need a filtering (post-processing) step, indeed possible elements of the intersection may not respect the original query.

However k-grams can also be used to help in spelling corrections.

Advantages of k-grams w.r.t. permuterm index are the efficient in terms of space and the fact that k-grams can help in correcting spelling mistakes.

Spelling corrections

The two main principles are: if a word is mispelled, then find the nearest one (this demands that we have a notion of nearness or proximity between a pair of queries), if two or more words are tied (ore nearly tied), select the most frequent word.The *isolated-term* correction attempts to correct a single query term at a time, even when we have multiple-term queries.

The idea behind the **Edit/Levenshtein distance** is that the distance between two words w_1 and w_2 is given by the smallest number of edit operations that must be performed to transform w_1 in w_2 (i.e. to make them equal). Possible edit operations are *insert* a character in a string, *delete* a character from a string, *replace* a character in a string.

This is a dynamic programming problem, there is a classical dynamic programming algorithm that runs in $O(|w_1| \times |w_2|)$, being $|\cdot|$ the length of the word.

Let $w_1 = v_1 \cdot a$, and $w_2 = v_2 \cdot b$, with v_1, v_2 words, a, b characters, then the distance $d(w_1, w_2)$ is the minimum between:

- $d(v_1, v_2) + 1$ if $a \neq b$ (we replace a by b);
- $d(v_1, , v_2)$ if $a = b$;
- $d(v_1, v_2 \cdot b) + 1$ we remove a from the first word;
- $d(v_1 \cdot a, v_2) + 1$ we add b in the second word.

Remark that the distance the word and an empty string is the length of the word, the distance between two empty strings is 0.

The complexity is given by the fact that we compute the elements of the matrix having as rows the letters in w_1 and columns the letter in w_2 (or vv, plus the empty string ε), each cell (i, j) is filled with the distance between the word composed by the first i characters of the first word and the one made by the first j characters of the second, hence the complexity in time (and space) is $O(|w_1| \cdot |w_2|)$.

Computing the edit distance on the entire dictionary can be too expensive, but we can use some heuristics to limit the number of words. We can for example use k-grams to retrieve terms with low edit distance from the misspelled word. We can try to retrieve terms with many k-grams in common with a word (with the hypothesis that this is indicative for a low edit distance, even if this is not always true).

We can use the **Jaccard coefficient** given by $\frac{|A \cap B|}{|A \cup B|}$, being A and B sets to select the terms obtained by looking at the k-gram in common. It is a measure of overlap. The two sets we consider are the set of k-grams in the query q and the set k-grams in a vocabulary term.

The fact is that sometimes all the words are spelled correctly inside the query, but one is actually the wrong word (e.g. "form" instead of "from"), in these cases we can substitute one at a time the words of the query with the most similar in the dictionary, perform the modified queries and look at the variants with most results. This can be expensive, but some heuristics can help.

Another kind of correction that we want is **phonetic correction**, i.e. when a word is written as it sounds (for Italian this does not happen, but for English and many other languages this is very common). In this case we want to look for words that *sounds* similar, not necessarily having similar letters. So a possible solutions is to put words in equivalence classes, where all words that sound similar are inside the same class. This is language dependent, for the English we have **Soundex** algorithm:

- the first letter is kept unchanged, usually it is not mispelled;
- change all the occurrences of A, E, I, O, U and Y with 0;
- we convert letters according to a specific table;

- remove all occurrences of 0 and pad the string with trailing 0s;
- return the first four positions.

At the end we get a sort of *phonetic hash* of the word, and we search for the words with the same phonetic hash as the ones in the query.

Optimisation of Boolean Queries

Sometimes order is important (the result is the same, since intersection is associative and commutative, but the time needed can be different).

The goal is to select the order that reduce the size of the intermediate results, but we don't know the size of the intersection, except for the fact that $|A \cap B| \leq min(|A|, |B|)$, so we can use $min(|A|, |B|)$ as estimation. We can use the same considerations for the union, using $|A| + |B|$ as estimate.

Ranked retrieval

Recall that the Boolean retrieval we only returned a set (all documents matching a Boolean query). In the case of large document collections, the resulting number of matching documents can far exceed the number a human user could possibly sift through. So it is essential for a search engine to rank-order the documents matching a query. A first way of ranking them is to split them according to some kind of structure.

Many documents are not completely unstructured text (most documents have additional structure), but have have at least some **metadata** (i.e. specific forms of data about a document, e.g. title, author, publisher etc..). Some of these metadata can be structured (*fields*), other might be zones, arbitrary are of free text (the possible values of a field should be thought of as finite).

Parametric Indexes

To allow for searching inside the fields we might want to build additional indexes, called **parametric indexes**. A parametric index can be thought as a standard index that only has information about a field (e.g. all dates). They are still indexes, so everything works as usual, in particular union and intersection. There is one parametric index for each field.

Zone indexes

Zones are similar to field, except the contents of a zone can be arbitrary free text (and it can be thought of as an arbitrary, unbounded amount of text).

We can have a separate (inverted) index for each zone (e.g. cat.title, cat.abstract, cat.author). Another possibility is to have a single posting list in which the zones

are part of the postings (we can reduce the size of the dictionary by encoding the zone in which a term occurs in the postings).

So we can index zones and fields inside the document to reach the query language and work in a specific field of the document. But we can consider that different parts in documents may have different importance. So we can find a way to order (i.e. rank) the documents according to where we find the different terms inside the document.

This idea of assigning different scores to different zones of the documents and ranking results according to where we find different terms is called **weighted zone scoring** (or **ranked boolean retrieval**).

Scoring function

Consider a pair (q, d) of a query q and a document d, a **scoring function** associates a value in $[0, 1]$ to each pair (q, d) (by computing a linear combination of *zone scores*, where each zone of the document contributes a boolean value). Higher scores are better.

Suppose that a document has l zones, each zone has a weight $g_i \in [0, 1]$ for $1 \leq i \leq l$. The weights sums to one, i.e. $\sum_{i=1}^{l} g_i = 1$.

Given a query q let s_i be defined as: $s_i = 1$ if q matches in zone i, 0 otherwise (boolean score denoting a match between q and the i-th zone). Actually s_i can also be defined to be any function that maps "how much" a query matches in the i-th zone (i.e. the presence of query terms in a zone). The weighted zone score is defined as $\sum_{i=1}^{l} g_i s_i$.

In order to set the weights we could ask to domain experts (but this is not always possible), or we can have users label documents relevant or not w.r.t. a query and trying to learn the weights using the training data.

With only two zones, site score is computed as $score(d, q) = g \cdot s_{title} + (1-g) \cdot s_{body}$ (with g constant). Since we know the queries and the real relevance of the documents in the training set, we can compute the output that a weight g would give.

If we decide that relevant is 1 and non-relevant is 0 we can compare the real score with the computed one and compute the error (not that we usually don't have an error 0, but we want to minimise it).

Actually zone scoring with weights is not the only way of weighting the documents.

TF-IDF Weighting

We now consider the fact that a document or zone that mentions a query term more often has more to do with that query and therefore should receive a higher

score. A plausible scoring mechanism is then to compute a score that is the sum, over the query terms, of the match scores between each query term and the document.

We may want to assign different weights depending on the term and the number of times a term is present in the document. This works well with *free-form* text queries (i.e. query typed freeform into the search interface, without any connecting search operators, such as boolean operators): for each term in the query we compute a *match score* and then we sum the score of all the terms.

The **term frequency** $tf_{t,d}$ simply counts the number of times a term t is inside a document d (this is a weight for the term, depending on the number of occurrences of the term in the document). The more the term is present inside a document, the more we consider the document relevant w.r.t. that term (however we are still not considering the order of the words).

This view of the documents is called the **bag of words** model, in which we consider a document as a bag of words and the ordering of the terms is *immaterial* (it is ignored) but the amount of occurrences is *material* (it is considered), in contrast with Boolean retrieval.

The fact is that, as we have already seen, the number of occurrences not always represents the importance of a term (e.g. stop words). Hence counting the frequencies alone is not enough. We need also to weight the different weights w.r.t. how relevant terms are for a document. Raw term frequency suffers from the fact that all terms are considered equally important when it comes to assessing relevancy on a query. In fact, certain terms have little or no discriminating power in determining relevance.

The idea is that rare words inside a collection counts more. So we scale the term frequency by weight of the term that will depend either on the **collection frequency** cf_t of the term t (total number of occurrences of the term t in the collection) or the **document frequency** df_t of the term t (number of documents inside the collection in which the term t is contained). Usually we consider the document frequency (which can distinguish e.g. if a single document with 1000 instances of a term in a collection of 1000 documents).

The document frequency increases for example with stop words, but we want to give more relevance to the terms that are rare (df_t is larger when we want the penalties to be larger), so we use the **inverse document frequency**: $idf_t = log\frac{N}{df_t}$, with N number of documents in the collection, df_t document frequency.

Hence, combining the definitions of term frequency and inverse document frequency, we compute the **tf-idf weighting** as $tf - idf_{t,d} = tf_{t,d} \times idf_t$. We answer queries by computing $tf - idf$ for all terms and them summing up.

Rare terms will have a large $tf - idf$, frequent terms that are present many times or a rare term is present only a few times the value is low, the lowest is when a very frequent term is present only a few times.

Summarising, the $tf - idf_{t,d}$ assigns to term t a weight in document d that is:

1. highest when t occurs many times within a small number of documents (thus leading high discriminating power to those documents);
2. lower when the term occurs fewer times in a document, or occurs in many documents (thus offering a less pronounced relevance signal);
3. lowest when the term occurs virtually in all documents.

Scoring a document

We can see a document as a vector with one component for each term in the dictionary and having as elements the $tf - idf_{t,d}$ of the term t in the document d. for dictionary terms that do not occur in a document, this weight is zero.

To score a document for a query q we can simply sum the $tf - idf_{t,d}$ values for all terms appearing in q, i.e. $Score(q, d) = \sum_{t \in q} tf - idf_{t,d}$. In this way we can have a score that is non-zero even if a term does not appear (the penalty will depend on which term is not present). We are moving toward the way of partial matching.

Variants of TF-IDF

Notice that with $tf - idf$ not all instances of a term inside a document carry the same weight (idea of *diminishing returns*). Another observation is that we might be interested in the frequency of a term relative to the other terms in the document.

The idea is that it is unlikely that twenty occurrences of a term in a document truly carry twenty times the significance of a single occurrence. A common modification is to use not the tf, but its logarithm.

We can use the **sublinear tf scaling**: we can scale the $tf_{t,d}$ value to have the influence of additional terms reduced, i.e. $wf_{t,d} = 1 + logtf_{t,d}$ if $tf_{t,d} > 0$, 0 otherwise. This new value can be replaced when $tf_{t,d}$ is used ($wf - idf_{t,d} = wf_{t,d} \times idf_t$).

Another idea is to use **tf Normalization**, so to scale the $tf_{t,d}$ value to be dependent on the maximum term frequency in the document $tf_{max}(d)$, i.e. $\frac{tf_{t,d}}{tf_{max}(d)}$ (we normalize the tf weights of all terms occurring in a document by the maximum tf in that document).

In some cases we may want to perform some kind of smoothing to these indexes because they can swing a lot, so we normalize according to the number of terms in the entire document: $\frac{tf_{t,d}}{\sum_{t' \in d} tf_{t',d}}$.