New Cluster Extractor Design

bam

July 24, 2007

We basically want some kind of *cluster data structure*, a *build structure method* and a *cluster look-up method*. We further want to distribute the build method, as the index is now distributed, and we want to optimise the look-up method, as this currently seems to be the time-consuming operation.

The look-up method should take a record or an id, and return a set of clusters (cluster names). This method should work for both 'known' and 'unknown' records, i.e. both records present in the index, which was used to build the cluster data structure, and completely new records.

Note that we have converted from Euclidian distance to *cosine similarity*. This distance measure is recommended when data is textual.

1 Cluster Data Structures

We want two cluster data structures: a centroid data structure and a vocabulary data structure. To look up clusters for a given record (document), we need to extract a document vector using the vocabulary, and then calculate distance or similarity to centroids.

1.1 Vocabulary

The vocabulary structure could consist of one or more maps: A synonym map, which maps all synonyms to the same word and a boost factor and document frequency map, which maps all known words to a boost factor and a document frequency (see appendix A). The synonym map would be nice to have, but is not necessary. Note however that it could also be used as a map from classification codes to words, or we could have both a classification-code-to-word-map, a number of dictionaries or 'translation-maps' and a synonym map. Note that stemming could also be implemented as a map.

The boost factor and document frequency map is the vocabulary, i.e. when constructing vectors only words from this map are considered. The default boost factor is one, but we imagine an increased boost factor to terms from specified classification fields.

Note that the terms do not necessarily have to be words, but can also be word combinations.

1.2 Centroids

The first centroid data structure suggestion was a simple set of centroids and diameters, similarity-thresholds or approximate sizes. The trouble with the simple set data structure is that to determine which clusters a document belongs to, we need to compare the document vector to all centroids, which is expensive. We should devise a method of determining, which centroids it is necessary to calculate distances to; or we should devise a smarter sparse vector or a smarter algorithm…

We will use the *dendogram*; see `en.wikipedia.org/wiki/Dendrogram`. The dendogram is a tree structure, which contains the calculated centroids (or clusters) as leafs and joined centroids as internal nodes. To construct the dendogram we need a join method, which joins 'close' centroids creating new 'joined centroids'. We already have a join method, which joins very close centroids as part of our clustering algorithm, and this method can be adjusted to help build the dendogram. Once we have the dendogram, we hopefully only need to compare the document vector to a logarithmic number of centroids…

2 Cluster Look-up Method

Given the suggested cluster data structure, we need a full record rather than an id to look up clusters. We should first decide a type. Choosing `dk.statsbiblioteket.summa.common.Record` will mean parsing content twice, which is expensive. Choosing `org.apache.lucene.index.TermFreqVector[]`, which is the return type of the `getTermFreqVectors(int docNumber)` method in `org.apache.lucene.index.IndexReader` is closest to the current cluster extractor implementation, but limits text analysis possibilities. We could also use `org.apache.lucene.document.Document` or invent a new 'intermediate' type, or we could use the document id and give the look-up method access to the index? We have decided to use Lucene `Document` as this can be used both as parameter and return type and can simply be enriched by the cluster provider method. Note that when the Lucene document is build it of course also contains the fields not to be stored; a document retrieved from index only contains stored fields.

The look-up method will translate the given record to a document vector using the vocabulary data structure. Distances from this vector to 'relevant' centroids will be calculated, and based on these the method decides which clusters the record belongs to. The centroid dendogram is used to decide 'relevant' centroids. The look-up has no effect on the cluster data structure. For a new record to affect the structure, a complete rebuild is required.

Assuming that the cluster data structures do not become huge, they can be copied to the index pc's and the cluster look-up service can be a local service.

3 Build Vocabulary

I suggest that we postpone the dictionary maps, and that when we have time, we look at WordNet (`http://wordnet.princeton.edu/`).

The boost factor and document frequency map is build by going through all terms in index. This can be done locally, and the local maps can be merged subsequently. The (lowercased) terms are matched against the "CCTerms" regex, and stop words ("CCNegativeTerms") are filtered out. The boost factor is determined based on new 'field-to-vocabulary-boost-factor' properties, and the document frequency is provided by the `org.apache.lucene.index.TermEnum.docFreq()` method. When merging, the maximum boost factor is chosen, and the document frquencies are added.

4 Build Centroids

The centroids are also build locally. This means that we find candidate terms and perform initial searches locally. Based on these search results, we build centroids and determine diameters or similarity thresholds and I think we should also determine expected sizes. These centroids sets are then moved to a central server and merged.

When merging centroids, closeness and names are checked, and in case the names are equal and/or the centroids are extremely close, the centroids are merged. We have a `joinCloseClusters` method which can be adjusted. As a second step, we should look at a split method. Both the join method and the split method should be employed globally, but could also be employed locally first…

When the centroids are merged, we will build a dendogram – a modified version of the join method can be used here.

5 Work Flow

We assume that there are local `ClusterBuilder` and `ClusterProvider` services on each 'index machine'and a central `ClusterMerger` service. It is assumed that the central `Score` module constructs both central and local services and makes sure that they are up and running (almost) all the time. The `Score` module also provides the configurations (properties) for the constructors.

TODO: come up with better service names

Note that the cluster builders have to be local as they need access to the local indexes. The cluster provider could be part of a 'preprocessor work flow' on a dedicated machine.

The build work flow is handled by either the central cluster merger or by a dedicated 'cluster work flow handler'. The configurations will also contain a 'build interval', i.e. build weekly or build every five days. The class responsible for the work flow can use a scheduler and this configuration. We need to move the different clusterdata structures between machines, and it is assumed that a service is provided for moving files between different machines. The work flow looks something like this:

- For each index machine:
  - run the `ClusterBuilder.buildVocabulary` method
  - move the created vocabulary data structure to a central location (defined in the properties).
- Central: Call the `ClusterMerger.mergeVocabularies` method.
- For each index machine:
  - copy the new full vocabulary to this machine
  - run the `ClusterBuilder.buildCentroids` method
  - copy the new local centroid data structure to a central location.
- Central: Call the `ClusterMerger.mergeCentroidSets` method.
- For each index machine:
  - copy the new full centroid datastructure to this machine
  - call the local `ClusterProvider` update method

6 Parameters, Return Types and Properties

Properties (`dk.statsbiblioteket.summa.common.configuration.Configuration`) are assumed to be given to the constructor of all classes to be implemented. For `ClusterBuilder` and `ClusterProvider` the path to the local index as well as paths to local and global data structures (that is local and global vocabulary maps and local and global centroid sets) are assumed to be properties.

This means that the build methods do not need any parameters, and they can be void methods (they can fail for different reasons, but this is probably better modelled by different exceptions). The provide method takes a Lucene `Document` parameter and returns a Lucene `Document` enriched with clusters (alternatively the return type could have been a dedicated 'ClusterResult' type).

We also have to decide on how to update. If the paths change, I think we should simply construct new services. However if the paths are the same, but the content has changed, we need to reload. This is a part of the build methods, but the `ClusterProvider` should probably have an update method.

The paths to the vocabulary maps and centroid sets used by the central `ClusterMerger` are also properties. Note that the merger needs a set of paths corresponding to the set of local structures. The merge methods can also be no parameter void methods.

A tf-idf

From Wikipedia, the free encyclopedia

The tf-idf weight (term frequency-inverse document frequency) is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. Variations of the tf-idf weighting scheme are often used by search engines as a central tool in scoring and ranking a document's relevance given a user query.

The term frequency in the given document is simply the number of times a given term appears in that document. This count is usually normalized to prevent a bias towards longer documents (which may have a higher term frequency regardless of the actual importance of that term in the document) to give a measure of the importance of the term ti within the particular document.

$$\mathrm{tf_i} = \frac{n_i}{\sum_k n_k}$$

where ni is the number of occurrences of the considered term, and the denominator is the number of occurrences of all terms.

The inverse document frequency is a measure of the general importance of the term (obtained by dividing the number of all documents by the number of documents containing the term, and then taking the logarithm of that quotient).

$$\mathrm{idf_i} = \log \frac{|D|}{|\{d : t_i \ni d\}|}$$

with

- $|D|$: total number of documents in the corpus
- $|\{d : t_i \ni d\}|$: number of documents where the term ti appears (that is $n_i \neq 0$).

Then

$$\mathrm{tfidf} = \mathrm{tf} \cdot \mathrm{idf}$$

A high weight in tf-idf is reached by a high term frequency (in the given document) and a low document frequency of the term in the whole collection of documents; the weights hence tends to filter out common terms.