MultiLingualSupport

Multi-Lingual Support in Nutch

Jérôme Charron

17 June 2005

DRAFT

Introduction

The goal of this proposal is to provide a solution for multi-lingual support in Nutch. Multi-lingual support means to be able to use a language specific Analyzer during searching and analysing.

Configuration

The configuration of this behaviour is done using the standard `plugin.includes` and `plugin.excludes` properties of the nutch configuration file. For instance, to add French, English and German analysis support for both document analysis and query parsing, just add in the Nutch configuration file:

```
<property>
  <name>plugin.includes</name>
  <value>...|analysis-(fr|en|de)|...</value>
  ...
</property>
```

If no language specific analyzer plugin is specified in the configuration then the default `NutchAnalyzer` implementation (ie `NutchDocumentAnalyzer` will be used for document analysis (similar to the actual implementation), and no language dependent analysis will be performed on the `Query`.

Nutch Analysis Plugin

NutchAnalyzer Extension Point

The `NutchAnalyzer` interface defines a new Nutch extension point. This interface is simply an abstract class that extends the Lucene `Analyzer` class, so that Lucene analyzers could be easily integrated as `NutchAnalyzer` plugins.
`NutchAnalyzer` extensions should define the attribute "`lang`" in order to be associated to a particular language code.

AnalyzerFactory

The `AnalyzerFactory` class is responsible of instanciating the `NutchAnalyzer` implementation to use depending on the Nutch analysis plugins configuration and a specified language code.

The `AnalyzerFactory` policy for finding the `NutchAnalyzer` extension to use is to return the first one that match a specified language code. If none is found, then the default `NutchDocumentAnalyzer` is returned.

Document Analysis

Language Code Source

The language specific document analysis is based on the result of the LanguageIdentifierPlugin. It just call the `AnalyzerFactory` with the `lang` attribute provided by the LanguageIdentifierPlugin.

Code modifications

Impacts on the actual code is very light to add multi-lingual capabilities for document analysis. First, on the part of the code of the `IndexSegment` class that add a document to the index, the line

```
indexWriter.addDocument(doc);
```

should be replaced by

```
indexWriter.addDocument(doc, AnalyzerFactory.get(doc.get("lang")));
```

so that, the `IndexWriter` is called with the good `Analyzer` implementation.

(Note by KurosakaTeruhiko) This seems to have been implemented in Nutch 0.8. The following lines are found in `Indexer`, not `IndexSegment` which no longer exists in Nutch 0.8:

```
final AnalyzerFactory factory = new AnalyzerFactory(job);
 .
 .
 .
NutchAnalyzer analyzer = factory.get(doc.get("lang"));
```

Second, the `NutchDocumentAnalyzer` class must implement the `NutchAnalyzer` class.
(Note by KurosakaTeruhiko) This has been done in Nutch 0.8.

Query Analysis

Language Code Source

The language specific query analysis is based on a `lang` attribute like for the Document Analysis. But the `lang` attribute in this case must be retrieved from the front-end using the following policy:

1. Use an optional `lang` attribute provided by the search interface. 2. If no such attribute is provided by the search interface, then uses the Browser language. 3. (try to identify the query language using the LanguageIdentifierPlugin)(Note by KurosakaTeruhiko: This probably won't work well because queries are usually too short to tell a language. "chat" can be English or French, for example. What language is "Euro"?)

Code modifications

The query analysis requires more code modifications than the document analysis. The first impact on the searcher code is to be able to get the `lang` attribute from the front-end. This is done by adding the following method in the `Query` class:

```
public static Query parse(String queryString, String lang) throws IOException;
```

This method then uses the `AnalyzerFactory` to retrieve the analyzer to use for parsing the query terms. The `NutchAnalysis` class that is only used as a query parser, will be renamed to `QueryParser` (like in Lucene), and will be very similar to the Lucene `QueryParser` by providing a `parse` method with an `Analyzer` as parameter:

```
public static Query parse(String query, Analyzer analyzer);
```

—
(Note by AlessandroGasparini) In Nutch 0.8.1 this is already implemented in the `Query` class:

```
public static Query parse(String queryString, String queryLang, Configuration conf)
throws IOException {
    return fixup(NutchAnalysis.parseQuery(
            queryString, AnalyzerFactory.get(conf).get(queryLang), conf), conf);
  }
```

More Notes

(Note by AlessandroGasparini) The specified analizers performs language Stemming so if you would you can search for "retrieval" and you can get all the documents in which the stem "retriev" appears. On my configuration i used the LanguageIdentifier, and the analysis-(language) plugins for both the crawler and the web application. I've attached my nutch-site.xml to show the complete plugin configuration nutch-site.xml

The use of stemming needs to be carefully evaluated for a given corpus and target audience. There are a few aspects to be considered:

- stemming is language-specific, but if the corpus is mixed-language, even if we detect the language of each document and apply a proper stemmer, we are still facing the problem of correct identification of the language of the query (and applying the correct stemmer to the query - see above).
- stemming is likely to increase recall, but also it's likely to reduce precision. It's more useful for morphologically rich languages, and it's also more useful for smaller collections (where users prefer to receive any results, even if they are less precise, over receiving none results whatsoever -

thus trading precision for recall). For larger corpora consisting of mostly mono-lingual documents stemming usually doesn't improve quality of results.
- there is usually more than one different stemmer implementation for a given language, each giving different results in terms of precision/recall for a given corpus. Sometimes an aggressive, iterative stemmer (such as the Porter stemmer) may give worse results than a light custom stemmer that only conflates single/plural forms.

References

- The "Multi-lingual support" thread on the nutch-dev mailing list.
- The "Indexing multiple languages" thread on the java-user Lucene mailing list.
- Stemming - Wikipedia, the free encyclopedia