**Brief Overview**

Apache Nutch is used to crawl links obtained from DMOZ data. The crawled data is pre-processed using Apache Lucene tokenizer and filters, resulting terms are used to build vector-space index. The application is setup as a Java web application using Apache Maven and packaged into a WAR file. Upon deployment, it is accessible at <http://localhost:8080/irproject/>. The application upon receiving a search query does the necessary pre-processing, then uses JWNL (Java WordNet library) to access the WordNet data for incorporating new terms for query expansion, then calculates the cosine similarity against the relevant documents and returns the sorted results to the user. The user is also provided the opportunity to submit relevance feedback for each of the relevant documents, which the application makes uses of in calculating both positive and negative feedback using Rocchio method, updates the query accordingly and generates new results to the user. There are two separate projects, the ‘IRProject’ which is the actual project that accepts queries and feedback against a collection of crawled web pages, and the second project ‘IRProjectEval’ which is cloned from the original project and edited to do the evaluation over test data. Both projects are accessible at following github links:

<https://github.com/blouke/IRProject>

<https://github.com/blouke/IRProjectEval>

The size of the source code for each project is rather small for each project around only 2MB. The above repositories also contains IRProject.war and IRProjectEval.war files respectively for each project ready to be deployed on Tomcat container. The IRProject makes use of WordNet by using JWNL (Java WordNet Library) which expects a properties.xml file to point to the actual absolute path to the WordNet directory. The properties.xml file in IRProject.war file needs to be edited, and the value attribute of the following element needs to point to the actual WordNet directory which may be different based on Tomcat directory.

<param name="dictionary\_path" value=”specify wordnet directory path here, it should end with …tomcat\webapps\irproject\wordnet\dict”/>

This path is system dependent and will be different for Linux. This properties file can be edited before deploying it to Tomcat by editing it in the war file, in the packaged war file the properties.xml is located at /WEB-INF/classes/properties.xml. The war file is just an archive file, it could be opened and edited by 7zip program.

**Preliminary steps**

1. DMOZ open directory project data dump file <http://rdf.dmoz.org/rdf/content.rdf.u8.gz> is downloaded for the purpose of collecting seed URLs. The format of the file is RDF-like and not strict RDF. A parsing program based on custom regex, consisting of files DMOZParser.java and RDFHandler.java contained in package ir.parser is used to parse DMOZ data. The output of the command line program is a file src/main/resources/links.txt that contains all the URLs from DMOZ data in relevance to the topic ‘Top/Shopping/Clothing/Casual’.
2. The file links.txt containing 296 seed URLs one per line is injected to Apache Nutch. File src/main/resources/nutch-commands.txt contains the list of commands and instructions employed to inject, fetch and parse URL content. Few decisions were made with respect to crawling in order to limit the collection of data set. Nutch property ‘db.ignore.external.links’ was set to ‘true’ in order to avoid accumulating URLs outside the original URLs domains. Only two rounds of Nutch fetch and parse commands were used after updating the crawl database which contains the links to crawl. A brief description follows.

Nutch 1.x Crawling Steps

# INJECT the seed URLs

bin/nutch inject crawl/crawldb urls

# FIRST ROUND

bin/nutch generate crawl/crawldb crawl/segments

export SEGMENT=`ls -d crawl/segments/2\* | tail -1`

bin/nutch fetch $SEGMENT

bin/nutch parse $SEGMENT

bin/nutch updatedb crawl/crawldb $SEGMENT

# SECOND ROUND

bin/nutch generate crawl/crawldb crawl/segments

export SEGMENT=`ls -d crawl/segments/2\* | tail -1`

bin/nutch fetch $SEGMENT

bin/nutch parse $SEGMENT

bin/nutch updatedb crawl/crawldb $SEGMENT

#merge the segments

bin/nutch mergesegs crawl/merge -dir crawl/segments/

#dump only parsed text of each url from all segments.

bin/nutch readseg -dump crawl/merge/\* crawl/dump -nocontent -nofetch -nogenerate -noparse –noparsedata

Note:

Set Nutch property fetcher.max.crawl.delay to 5 seconds to ignore pages beyond this much delay.

1. The execution of Nutch commands above results in a Hadoop SequenceFile which is fed into another command line parsing program NutchSegmentParser.java contained in package ir.parser. The output is a text file src/main/recources/dump which contains the URLs along with their content which still needs to be parsed to extract URLs and content in a meaningful format.

**Instructions for Application deployment.**

The main application is created as a java web application and packaged into a WAR file. The application makes use of Java Servlets to serve HTTP requests for searching queries, which requires the application to be deployed on any Java Servlet container or an JEE Application server. Tomcat was chosen for testing the application during development. Following instructions pertain to Tomcat and $TOMCAT\_HOME refers to the Tomcat installation directory. In order to deploy the application to Tomcat, simply copy the WAR file into the $TOMCAT\_HOME/webapps/ folder and start Tomcat by invoking $TOMCAT\_HOME/bin/startup.bat (for Microsoft Windows). Point the browser to <http://localhost:8080/irproject/> to start using the application. Tomcat unpacks the WAR file into a directory by the same name resulting in $TOMCAT\_HOME/webapps/irproject/, this directory will contain the dictionary and the vector-space Index in a Java serialized binary format file named ‘index.ser’. Upon initial deployment of the application, the Controller servlet SearchController.java in package ir.controller builds the index and persist it to the disk as index.ser which may take about a minute. Any subsequent invocation of the application will make use of the index.ser rather than creating the index again from scratch. In order to force the application to re-create the index, stop Tomcat and delete the file $TOMCAT\_HOME/webapps/index.ser.

**Navigating the Application**

The application contains two publicly accessible JSP pages index.jsp and result.jsp along with a stylesheet at src/main/webapp/ folder. The index.jsp is the main landing page where search queries are to be submitted. The results of the search query are presented in result.jsp. The result.jsp presents the list of all relevant documents along with a checkbox to specify the relevance of particular documents in order to provide relevance feedback to the application which uses Rocchio method to accommodate both positive and negative relevance feedback.

**Application Description**

The application is build using Apache Maven build utility. The pom.xml file contains all the dependencies. The source code is divided among four packages.

**Parsing**

Source code package ir.parser contains two sets of classes meant for parsing data, both are invoked as command line java classes.

**Application Front Controller that serves HTTP requests**

Source code package ir.controller contains only one file SearchController.java, a servlet which intercepts all HTTP requests and runs the query against the index by passing the query to the QueryProcessor.java in package ir.query. SearchController is responsible for creating the index and serializing/deserializing it along with other initializations and also for routing the HTTP requests to the appropriate view, the JSP page.

**Query Processing**

Source code package ir.query contains the class QueryProcessor.java which is initialized with a reference to the index and also with WordNet. Upon receiving search query, it invokes method processQuery which makes use of Apache Lucene Tokenizer and StopFilter for stop word removal. Then JWNL (Java WordNet library) is used along with WordNet data to expand on query terms. Word sense disambiguation is not accommodated for, so words corresponding to SynSets of all word senses obtained from WordNet are incorporated into the query vector. The query vector is then used to calculate the cosine similarity against the set of relevant documents. The set of relevant documents is then sorted and returned to the controller serlvet which forwards the collection of relevant documents to the result.jsp view.

The QueryProcessor class also contains a method generateUpdatedResults that receives a list of relevant documents and their relevance feedback submitted by the user, it then employs Rocchio method with beta value of 0.5 for positive feedback and gamma value of 0.1 for negative feedback. The original query vector is then augmented with the results of the Rocchio method. The new query is then used to calculate cosine similarity against a new set of relevant documents, which are then sorted and returned to the controller servlet which forwards the collection of relevant documents to the result.jsp view.

**Indexing**

Source code package ir.indexer

TermIndexer.java makes use of other classes in this package to build the index. It starts by parsing the Nutch output data by using a custom regex. Lucene tokenizer and filter classes are used for processing the contents of the crawled data, Porter stemmer is used for stemming the terms. All classes in this package implements the Serializable interface so that the index can be serialized to the disk.

**Evaluation**

The test data used for system evaluation is accessible at <http://web.eecs.utk.edu/research/lsi/corpa.html>. The test data contains 423 Times Magazine Articles (from 1963). The evaluation is done in a separate project named ‘IRProjectEval’. This project can be run to generate evaluation results, but an html document generated by the system ‘evaluation\_results.html’ containing the evaluation results is also included in the submission. The evaluation is done by retrieving relevant documents by running each test data query against the system. Multiple similarity measures are used to measure system performance namely Cosine, Jaccard and Dice similarity measures. The evaluation page result.jsp displays a table containing precision, recall and f1-measure for each query in the test data along with the number of documents retrieved by the system and the actual test data judgments. The table displays only the results using the cosine similarity measure because it produced the better results among all the similarity measures used. The evaluation page also displays Mean Absolute Error for all three similarity measures which are:

**Mean Absoulte Error**

Cosine similarity :2.892

Dice similarity :5.361

Jaccard similarity :8.904