# Lucene-Based Long Query Reductor – Reference Manual

December/2017

## 1 Overview

This project implements a classification-based query reduction system. Once it is used a Part-Of-Speech tagging process to extract features, it is assumed that the query to reduce is sytactically structured, and high performance is not guaranteed otherwise.

This system consists on three main components:

- the Part-Of-Speech Tagger, which defines the syntactic structure of the original query;
- the Data Treating module, that is responsible for extracting the features used for classification. This part of the system deals with both the query to reduce and the building of training data from text.
- the Classifier, which receives a set of features for each word of the query, and, based on a dataset generated by the **Data Treater**, classifies if it should be removed or not.

The code is fully available at `https://github.com/mowrilo/query-processing`. This document provides descriptions of each of the methods developed, its inputs and outputs. Inside the repository there is an example code of how to reduce a query.

## 2 POSTagger

The Part-Of-Speech tagger is implemented using a Markov Model. It has the following variables:

- countWordTag: Hash map which stores the number of times a given word is observed under a given tag.
- countTagTag: Hash map which stores the number of times a given tag is observed after another tag.

- sumOfTags: Hash table that stores how many words are observed under each tag.
- sumOfWords: Hash table that stores how many times each word is observed on the training corpus.
- tags: Vector that stores all of the unique observed tags.
- nWords: integer value that stores the vocabulary size.

The POS Tagging process consists of the following methods:

## train

### Input

The path to the training corpus.

### Output

Nothing (void).

### Operating mode

The training process reads every file on the training corpus path. On each file it is read each line as a sentence and it is counted each occurence of tag under a certain word, as well as each occurence of tag after a certain tag. These counts are used to compute probabilities on the tag process.

## addToWordMap

### Input

A String consisting on a word and a String consisting on a tag.

### Output

Nothing (void).

### Operating mode

The method increments the Word-Tag count Hash on the position of the given word and tag. If the word or the tag is not present, it adds their key to the Hash.

## addToTagMap

### Input

Two strings consisting on two tags observed in sequence.

#### Output

Nothing (void).

#### Operating mode

The method increments the Tag-Tag count Hash on the position of the given tags. If one of the tags is not present, it adds their key to the Hash.

### saveModel

#### Input

A string consisting on the complete path to the file to save the POS model.

#### Output

Nothing (void).

#### Operating mode

This method saves on a file all of the class' variables described above, such as the occurence counts.

### loadModel

#### Input

A string consisting on the complete path to the file containing the classes' variables described above.

#### Output

Nothing (void).

#### Operating mode

This method reads the file and updates the classes' variables.

### tag

#### Input

A string consisting on a sentence to be tagged.

#### Output

A string consisting on the tags predicted for the input sentence.

#### Operating mode

The tag process uses the Viterbi algorithm. It is created a special tag for beginning and end of sentence. Then, the algorithm iterates over all of the words on the sentence, computing the probabilities of the word occuring under each of the possible tags. These probabilities are computed based on the probabilities of the previous word and the probability of the current word occuring under a tag. It is then achieved the sequence of tags with the greatest probability of occuring.

It is used the log-probability for numeric reasons.

### saveSumOfWords

#### Input

Nothing.

#### Output

Nothing (void).

#### Operating mode

This method uses the Word-Tag count Hash to calculate the number of times which word occurs on the training corpus.

### saveSumOfTags

#### Input

Nothing.

#### Output

Nothing (void).

#### Operating mode

This method uses the Tag-Tag count Hash to calculate the number of words that occur under each tag.

### getProbWordTag

#### Input

Two strings, consisting on a word and a tag.

#### Output

A double value, consisting on the probability of the given word occuring with the given tag.

#### Operating mode

This process uses a smoothing parameter $\lambda$ to deal with previously unseen words. The probability is then a combination of the number of times the word is observed under the tag and the total number of words in which each tag is observed.

### getProbTagTag

#### Input

Two strings, consisting on two tags that occurred in sequence.

#### Output

A double value, consisting on the probability of the second tag occuring after the first tag.

#### Operating mode

The probability is simply the count of the number of times the latter tag occurred after the former tag, divided by the total count the latter tag occurred.

## 3 DataTreater

The DataTreater class does the work of extracting features of words, and thus structuring unstructured text. Is has the following variables:

- data: this is a list of lists, where each list represents a word, being a set of features.
- pos: this is a simple POS Tagger, to use for feature extraction.
- features: this is an array of Strings, which stores the selected POS features.
- maxIdf, maxTf, maxPos, minPos, minLen, maxLen: maximum and minimum values of the continuous features, used for normalization.

The class has the following methods:

### buildData

A String consisting on the path to the training text and an IndexReader from Lucene.

#### Output

Nothing (void).

#### Operating mode

This method iterates through all the files inside the path given, reading its text and building a training dataset. For each file, the target reduced query must be with a *<title>* tag on the beginning of its line, while the original long query comes two lines after it. The method does some changes to the string, such as putting whitespace next to punctuation to separate it from words and removing periods that are not on the end of the sentence. It then tags the sentence using the PosTagger.

Then, it uses the IndexReader to extract collection statistics such as document frequency and total term frequency, using them to build a feature vector to each word. Only the tags present on the *feature* array are added to the vector.

The vectors are then stored on a list, updating the class variable *data*. The index data must have been stored under the *"contents"* field.

### saveData

#### Input

A String that consists on the complete path of a file to save the training data.

#### Output

Nothing (void).

#### Operating mode

This function simply uses the serialization method from Java to store the *data* variable to a file specified by the user.

### loadData

#### Input

A String containing the file to load the data from.

#### Output

Nothing (void).

#### Operating mode

This method just reads the data file and updates the classes' variables.

### analyzeSentence

#### Input

A String containing a sentence to analyze and an IndexReader from Lucene.

#### Output

An ArrayList of ArrayLists, where each list is a feature vector for each word on the input sentence.

#### Operating mode

This method makes the feature extraction the exactly same way the *buildIndex does*, structuring the data and returning an ArrayList of ArrayLists, representing the features for each word.

### buildCSV

#### Input

A String consisting on the path to the training sentences, an IndexReader from Lucene and another String, consisting on the complete path to a destination CSV file.

#### Output

Nothing (void).

#### Operating mode

This method makes the feature extraction the same way the *buildData* does, with a difference: it saves all of the POS Tag features under three discrete features, representing the current word Tag, the former and the next Tags. The structured data is then saved under a CSV file specified by the user.

## 4 LogisticRegressionClassifier

This is one of the classifiers implemented. It calculates a probability value, between 0 and 1, that a word has to be removed. Using a predefined threshold, it can return the discrete values 0 or 1.

It is trained using the Maximum Likelihood method, maximizing the Likelihood function using gradient descent.

It has the following variables:

- data: the data built by the *DataTreater*
- params: a list of double values, the parameters of the logistic regression. For $N$ features, there are $N + 1$ parameters.

- maxIdf, maxPos, maxTf, minPos, maxLen, minLen: max and min statistics to be used for data scaling.

The classifier has the following methods:

## logisticFunction

### Input

A double value.

### Output

Another double value.

### Operating mode

This function just calculates the logistic function on the input:

$$logistic(x) = \frac{1}{1 + e^{-x}}$$

## gradient

### Input

Nothing.

### Output

An ArraList of double values.

### Operating mode

This function calculates the analytic gradient of the Likelihood function for each of the $N + 1$ parameters, and returns a vector of $N + 1$ elements.

## loadData

### Input

A String containing the path to a file to load. This file has to have the data generated by the *buildData* method from the *DataTreater*.

### Output

Nothing (void).

#### Operating mode

This method just loads the data on the file and updates the classes' variables.

### loadModel

#### Input

A String containing the path to a file containing the parameters of the Logistic Regression.

#### Output

Nothing (void).

#### Operating mode

This function just reads the file where the parameters were saved using *LogisticRegressionClassifier.saveModel()*.

### saveModel

#### Input

A String containing the complete path to a file where the current model parameters will be saved.

#### Output

Nothing (void).

#### Operating mode

This method just serializes the data with the ObjectOutputStream from Java and saves it to the file specified by the user.

### normalizeData

#### Input

Nothing.

#### Output

Nothing (void).

#### Operating mode

This method scales the continuous features from the data, using the max and min values stored.

## getMaxLogLik

### Input

Nothing.

### Output

A double value consisting on the Log-Likelihood value on the current data and parameters.

### Operating mode

This function computes the Log-likelihood value on the current data and model parameters according to the following equation:

$$LogLik = \sum_{x \in C=1} log(logistic(x))) + \sum_{x \in C=0} log(1 - logistic(x)))$$

## initParams

### Input

The number of parameters of the model, consisting on the number of features plus 1.

### Output

Nothing (void).

### Operating mode

This method just initializes all of the model parameters as a low, fixed value.

## train

### Input

A string containing the path to a data file, an integer with the maximum number of iterations, and a double with the step of the learning process.

### Output

Nothing (void).

### Operating mode

This method uses Gradient Descent to find the model parameters that maximize the Log-Likelihood function, meaning it maximizes the odds of observing the current training data.

### predict

#### Input

An ArrayList consisting on a sample (word) and a double value with the classification threshold.

#### Output

An integer $C \in 0, 1$, consisting on which the word represented by the input sample will be removed or not.

#### Operating mode

This method computes the dot product of the model parameters and the sample, and then feeds this product to a logistic function. It then applies the threshold, returning 1 if the returned value is greater than it.

### predictAll

#### Input

An ArrayList of ArrayList, consisting on a list of samples (words) to classify and a double value consisting on the threshold.

#### Output

An ArrayList of integer values, consisting on the predicted values for each sample received.

#### Operating mode

This method just iterates over the list of samples and classifies each of them.

## 5 KNNClassifier

The K Nearest Neighbors is the other option of classifier. It is a very scalable and simple model, since it does not require the training step nor tuning parameters. Given a sample to classify, it just computes the distance to all of the training samples and assigns to it the most frequent label in the $K$ nearest training samples.

It contains the following variables:

- data: the training samples, as structured by the *DataTreater* module.
- k: the number of nearest neighbors the system considers to label an unknown sample.

- maxIdf, maxTf, maxPos, minPos, maxLen, minLen: maximum and minimum values of the continuous features, used for scaling.

It contains the following methods:

## normalizeData

### Input

Nothing.

### Output

Nothing (void).

### Operating mode

This method uses the max and min values to scale the continuous data between 0 and 1.

## loadData

### Input

A string containing the complete path to a data file, generated by the *DataTreater* module.

### Output

Nothing (void).

### Operating mode

This method just reads the objects saved on the data file and updates the classes' parameters.

## getEuclideanDistance

### Input

Two ArrayLists consisting on samples (words).

### Output

A double value consisting on the Euclidean distance between the two vectors.

#### Operating mode

This method just calculates the distance between the two samples by the Euclidean method:

$$D(sample_1, sample_2) = \sqrt{\sum_{i=1}^{N_{features}} (sample_{1i} - sample_{2i})^2}$$

### predict

#### Input

An ArrayList of double values, consisting of a sample representing a word.

#### Output

A class value $C \in 0, 1$, telling if the word should be or not removed from the query.

#### Operating mode

This method uses a class **Pair** as an auxiliary structure. The Pair class holds two values: one double and one integer. The distance of the input sample to all of the training samples is computed, and each distance is stored on a Pair instance. The distance is stored on the double value and the index of the training sample is stored on the integer value. Then, all of the Pairs are sorted by distance in ascending order. The $K$ first are then selected, and the input sample is classified as '1' if more than half of the $K$ nearest holds the label '1', and '0' otherwise.

### predictAll

#### Input

An ArrayList of ArrayLists, consisting on a list of several samples to classify.

#### Output

An ArrayList of integers, containing the class of each of the input samples.

#### Operating mode

This method just iterates over the samples provided and classifies each one of them using the *KNNClassifier.predict()* method.

### PairComparator

This auxiliary class just implements a comparator to sort Pair instances by the double value, i.e. the distance between samples.