**Information Retrieval And Web Search – Home Assignment 3**

**Hodaya Shalom 205858186, Michal Abramov 301834297**

**Abstract**

This report includes the decision we made in this assignment.

We chose to use Wekalibrary and Snowball library integrated with our implementations.

**Experiment Description**

We preformed our experiment as follows:

1. Reading the collection of documents and converting them to Weka format.

(note: files need to be written in Weka format only once, after they are written the first time this part can be skipped by specifying ‘recreateWekaDataFolders=false’ in ‘parameters.txt’)

1. Building classifier using the train set.
2. Testing the classifier using the test set
3. Printing results to output file

This experiment was preformed using different k values to choose the value that gives us the best results.

**Reading the data and writing documents files**

In order for us to use the Weka library we had to read the data and write each document to a file.

Directories hierarchy:

* Train and test directory
* In each train\test directory there is a directory for each class
* In each class directory there are files, each file contains the content of a document that belongs to that class.

**Building the classifier**

We load all of the files we wrote in the previous step.

To create the classifier we used Weka’s filters, classifiers, Multifilter and FilteredClassifier.

TF-IDF: In order to represent the documents as their tf-idf vectors we used Weka’s *StringToWordVector* filter, setting *setTFTransform* and *setIDFTransform* to true*.*

Lower Cases: we used the *StringToWordVector* filter to lower the case of all tokens by setting *setLowerCaseTokens* to true.

Stemming: we used the Snowball library to perform stemming. Snowball is an open source small string processing language designed for creating stemming algorithms. We used the porter stemmer algorithm that is included in the snowball library. To use this stemmer we set *StringToWordVector*’s *setStemmer* with an instance of Snowball.

For the classifier we used *IBK* set to work with the KNN algorithm.

We used multifilter and FilteredClassifier to combine the *StringToWordVector* filter and the *IBK* classifier.

We used multifilter and FilteredClassifier to run multiple filters which will run consecutively without changing the data structure. This enabled us to keep the documents names, which we needed for writing the results to the output file.

KNN: we used normalization and the k value read from the configuration.

We used normalization because we didn’t want the length of the document to have an effect on the classification.

After creating the classifier object, we build the classifier itself (classifier.buildClassifier) using the train data (training the classifier).

**Testing the classifier**

We test the classifier by classifying the test set and calculating micro f-score and macro f-score.

We ran each test with a different k value.

**Experiment results**

Coming in to the experiment we expected to get the worst results with k=1 and to improve the results as we increase the k value, but the results we got where opposite – we got the best results using k=1 and the results worsen as we increased the k value.

We also expected to see more variance between the micro f-score and macro f-score for each k value, but the results were very similar.

**Results:**

|  |  |  |
| --- | --- | --- |
| K | micro f-score | macro f-score |
| 1 | 0.885802 | 0.884994 |
| 5 | 0.874761 | 0.874269 |
| 10 | 0.858211 | 0.857485 |
| 20 | 0.831882 | 0.831526 |
| 30 | 0.809997 | 0.810377 |

et

**Configuration**

We’ve added 3 parameters to the parameters file:

* trainFolderWeka – the folder we write the train Weka files to
* testFolderWeka - the folder we write the test Weka files to
* recreateWekaDataFolders – true\false. To save the time it takes to write to Weka files, if we’ve already written the Weka files we can choose to skip this part (by configuring ‘false’).

**Testing and retrospect**

Running the code on the full data takes a very long time. This caused us issues in several occasions; whenever we wanted to test changes or run multiple k values.

We took several approaches of dealing with this issue:

* first, we ran the code on a sample train set and a sample test set. The smaller sets expedited the run time, which allowed us to test the code.
* Creating the Weka files once. As we mentioned in the previous section, we’ve added a parameter specifying to skip the part of writing the Weka files (in case we already wrote them in previous run). This also saved us a lot of run time.
* In order to run the code on multiple k values without reloading a data for each k value, we’ve added a function that return a list of k values to run. In ‘production’ mode it only returns the k we get as a parameter, but in ‘test’ mode it return a list of k values we want to test. This saves the time of loading the data for each k value, by loading it only once for all k values.

These time constraints (loading the data took about an hour, running each k took about 5 hours) prevented us from running the number of tests we would have liked to preform, in terms of the number of k values we wanted to run, why the results came out differently from what we expected etc.

Another issue we came across is losing the document ID when filtering the data. We solved this issue by using Multifilter and FilteredClassifier as we explained in the *Experiment Description*

Section.