

CSE 5243 Lab-1 Report

Harsh Gupta (gupta.749), Dhanvi Athmakuri (athmakuri.1)

September 20th 2016

1 Problem Statement

The purpose of this lab assignment is to *pre-process* the *reuters* dataset, to make it ready for feeding into data mining algorithms.

2 Implementation Details

Project Participants : This implementation has been accomplished by the collaboration of Dhanvi Athmakuri (athmakuri.1) and Harsh Gupta (gupta.749).

Participant Contributions : The layout for the code was discussed together. Dhanvi Athmakuri wrote the initial script to parse the files and extract the words and the word count files. Harsh Gupta implemented tf-idf, modularized the code and implemented command line input. The feature vector files were generated together. Dhanvi and Harsh together optimized the data structures later.

Implementation language and libraries used: We have used *python-2.7* programming language to implement our pre-processing methodology. The following libraries have been used to help process the files :

- `nltk` - (Natural Language Took Kit).
- `BeautifulSoup4` - provides the html parser to parse the documents.
- `SnowballStemmer` - to remove morphological affixes from words.
- `RegexTokenizer` - to extract the words that match a particular pattern.

Other libraries like `urllib2`, `sys`, `csv`, `time`, `math`, `collections` have been used for utility functions.

2.1 Assumptions made in the implementation

Our program can generate up to five different files, depending upon the input given by the user. We have used the *titles* of the articles and the collection of all the *words* in the documents, to generate these output files. We have used the information in the *topics* tag of every article, as the *class labels*. Articles or documents associated with multiple topics, will have multiple class labels (this is subject to change).

We have considered the title words to be separate from the document words as the titles are more important in identifying the type of the document. The format of the files generated and how to interpret them is explained in section. We will refer to the file formed from the titles of the articles as *title files* and the file formed from the body of the articles as the *bag of words file*. In generating the *bag of words file*, we have ignored the files with no body content.

3 Processing Methodology

Following is the step-by-step procedure which we have used to process the documents. Below every individual point of implementation, we have included the line(s) of code or the method (from our code), that accomplishes the task under consideration.

- 1. read the files from `http://web.cse.ohio-state.edu/~srini/674/public/reuters/`. We are reading the files directly from the internet.
- 2. We use the `html` parser from the `BeautifulSoup` library, to parse the files. Specifically, we read all the contents within the `reuters` tag, in each of the 21 files. The resulting 1000 articles are stored.
`unprocessed_files = BeautifulSoup(f,"html.parser")`
`self.parser.parse(unprocessed_files)`

3. From each such article we use the `RegexTokenizer` library, to extract only the words formed from the characters from the english alphabet. This step removes all the punctuation marks and other special non-alphabetic characters, including numbers. These set of words are further processed using the `stopwords` corpus of the `nlTK` library, to remove the stopwords.


```
tokenizer = RegexTokenizer(r'[A-Za-z]+')
```

`doc_words = tokenizer.tokenize(doc)`
 where, `doc` is the body of one single document (article). The snippet for stop word removal is included in the next point.
4. Further processing is done to remove morphological affixes from words. This is accomplished by using the `SnowballStemmer` library. We also consider the words of length lesser than 3, to be insignificant and remove them.


```
def clean_stopword_stem(self, doc)
```

 This method removes the stopwords and also stems them.
5. In exactly the same fashion, we extract the titles from each article and process the words in the title through the same process as above.
6. After extracting the relevant words, from the article, we count the number of times each word occurs in the article and update the information in the appropriate data structure (please refer to the code for appropriate details). Similarly, for words in titles, we record the number of times each word in the title occurs in that particular title.
7. we also maintain data structures to keep a track of the number of times a word occurs across the entire corpus and the number of documents it occurs in. These data structures will be useful while creating the **tf-idf** measure for each word.
8. In addition to extracting the words in the body and the title, we also extract the *topics* tags to use them as *class labels*. For articles which have an empty *topics* tag, we assign the class label as NOT AVAILABLE.
9. The following files are created to store essential information about the documents.
 1. `document_file.csv` : This file has a single row for each document or article. The row is in the following format :


```
document id, word:word count, word:word count,...,|,class labels
```

 Here, word count is the number of times the word occurs in the document.
 The `title_file.csv` stores the information in the exact same format, only that the words under consideration would be those in the title of the document.

4 Feature Vectors

The files - `document_file.csv` and `title_file.csv` are then used to generate the feature vector matrices. We have generated three feature vector matrices :

- `sparse_doc_file.csv` : feature vector with document words as columns and document as the row. Each value in the file represents the raw frequency of the word in the document. The columns are the set of all distinct words and the rows represent all the document across the entire corpus.
- `sparse_title_file.csv` : Exactly the same as the above feature vector, but instead of using the document words, here we use the title words. the columns represent the set of distinct words occurring in all the titles across the entire corpus and the rows are the document identification number.
- `sparse_tfidf_file.csv` : This file is the `sparse_doc_file.csv`, with each cell multiplied with the **tf-idf** measure of the word corresponding to that column.

4.1 tf-idf calculation

To calculate the **tf-idf** score for each word, we have used the following definition of idf :

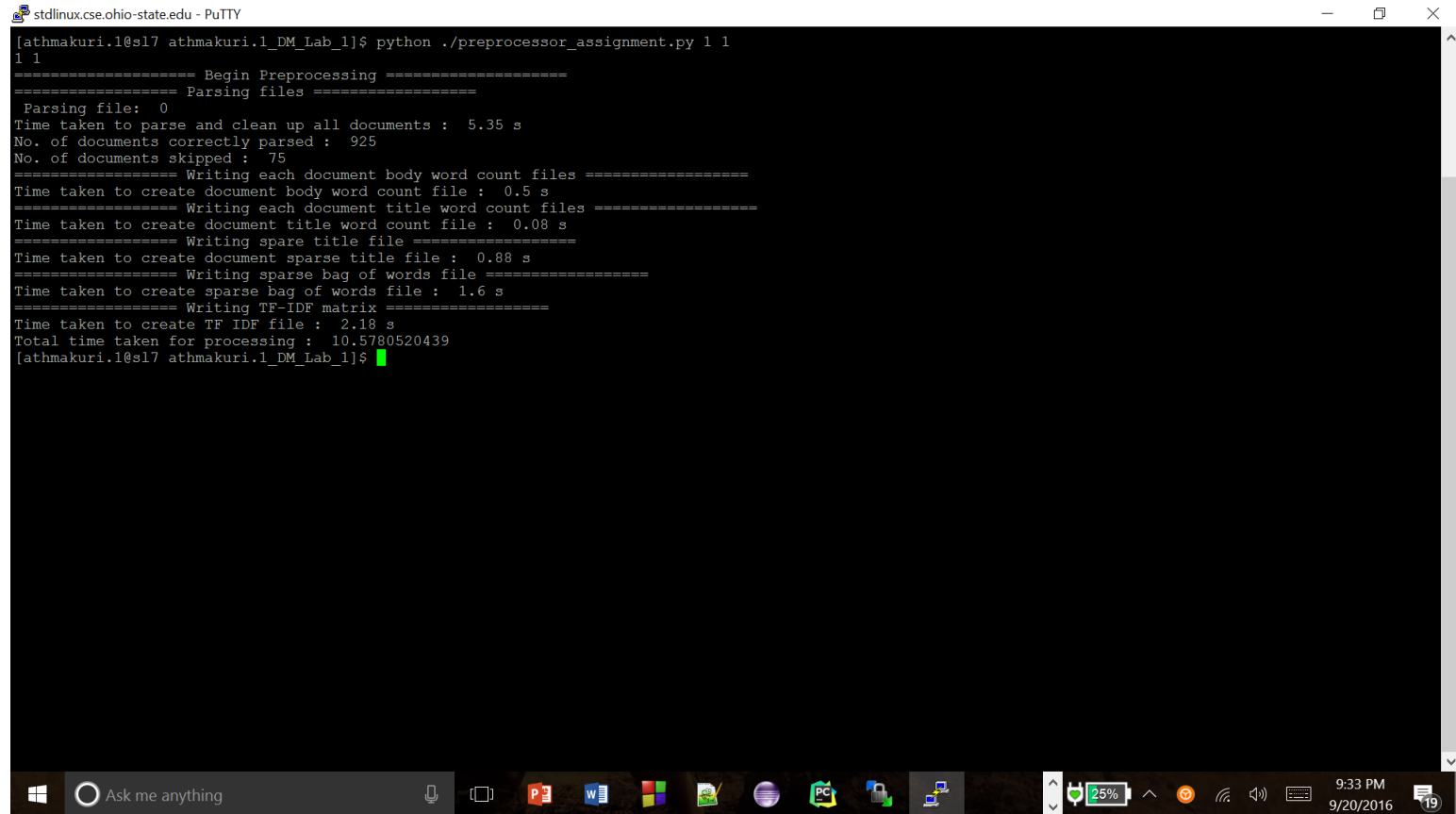
$$idf(word) = \log \frac{(total\ number\ of\ documents\ that\ are\ parsed)}{(total\ number\ of\ documents\ in\ which\ that\ word\ has\ occurred)}$$

The tf (term frequency) of every word is taken to be the raw-frequency of the word in the document.

5 Difficulties Encountered

Processing Time : The data set had 21k files to process and initially due to poor data structure and a lot of writing operations in files the running time was higher but then gradually we started using optimal data structure to the best of our

Figure 1: Sample Execution of the code for 1000 articles
DOCUMENTS/imagefile.png



```
stdlinux.cse.ohio-state.edu - PuTTY
[athmakuri.1@sl7 athmakuri.1_DM_Lab_1]$ python ./preprocessor_assignment.py 1 1
1 1
===== Begin Preprocessing =====
===== Parsing files =====
Parsing file: 0
Time taken to parse and clean up all documents : 5.35 s
No. of documents correctly parsed : 925
No. of documents skipped : 75
===== Writing each document body word count files =====
Time taken to create document body word count file : 0.5 s
===== Writing each document title word count files =====
Time taken to create document title word count file : 0.08 s
===== Writing sparse title file =====
Time taken to create document sparse title file : 0.88 s
===== Writing sparse bag of words file =====
Time taken to create sparse bag of words file : 1.6 s
===== Writing TF-IDF matrix =====
Time taken to create TF IDF file : 2.18 s
Total time taken for processing : 10.5780520439
[athmakuri.1@sl7 athmakuri.1_DM_Lab_1]$
```

knowledge and reduced running time drastically. We changed from using list to using map.

Incomplete or Empty Tags : Few files didn't have content in their body and topic tags. So it was a problems for different methods in there return type.

Parsing the file : We used `find_all` which returned the contents as a single string, thus making it difficult to retrieve the content of the children tags. We then changed to using `findAll()` which returned `bs4.element.Tag` making it easier to return thr content of the children tag.

To keep the report short, we have omitted describing the details of the data structures, methods and the class structure. The code documentation explains in detail, the variables used, the methods defined and the class structure.

6 Results

The number of files and the type of files generated are described earlier. Out of the five files that are generated the files of significant importance are `sparse_doc_file.csv`, `sparse_title_file.csv` and `sparse_tfidf_file.csv`. These files contain the feature vectors which can be given as inputs to data mining algorithms. These files are too big (of the order of 1.4GB), to be opened. Therefore, we have included 'sample' output files, which have been generated for only 1000 articles. These files have the same name prefixed with `sample_`. Please open those files to understand the output format. The above figure shows the screen shot of the sample execution of the code for 1000 articles, executed in stdlinux. As can be seen from the picture, Our code displays the number of documents parsed, number of them skipped and the time taken for each phase of the program to execute. The time taken for 1000 articles (one single .sgm) files is 10.57 seconds, as the screen shots shows. The average time taken by our program to genarate all the relevant output files for the entire corpus, is 250 seconds. This is subject to change with each run.

Please read the README file for instructions on how to execute the program. All the output files are present in `/home/8/athmakuri.1/athmakuri.1_DM_Lab_1` folder.