Data Mining COMP6237 – Individual coursework

Nicola Vitale
MSc Data Science
nv5g15@soton.ac.uk

ABSTRACT

In this paper the experience of understanding unstructured data is reported. The coursework, as part of the Data Mining module, involved the understanding of a series of texts data using various techniques.

1. INTRODUCTION

The data provided consisted of a corpus of 24 books. For each book we have the output of OCR processing: the content of each page has been provided as an html file.
The approach used involved two main phases:

- Data preprocessing
- Understanding the data

In the first phase the effort was aimed at obtaining the raw data by extracting the text and finding an appropriate data structure to work with Python. In the second phase various data mining techniques have been applied to the data corpus in order to understand its hidden structure.

2. DATA PREPROCESSING

Initially the text of the books has been extracted from the html files. For this purpose, it has been convenient to use the ***os library*** to iterate through the files and the ***re library*** to remove html tags keeping the text of each page. A regular expression to identify each tag has been found and once the page was cleaned the text of the book has been recreated and saved as a txt file. The output of this operation consisted in 24 txt files corresponding to the books texts (Figure 1).

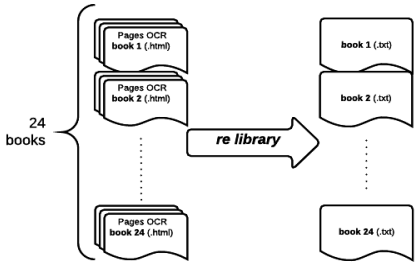

Figure 1: Texts collections

Then it has been decided to work with two main lists in Python: a list of 24 texts and a parallel list with the corresponding titles. These data structures have been saved using ***pickle library***. In general, over the project, this library has proved to be very useful since it allowed to save the output of phase and reassigning it to a variable (Figure 2).

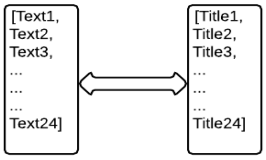

Figure 2: Python lists

3. UNDERSTANDING DATA

3.1 Feature extraction and VSM

The approach followed to mine the texts is the Vector Space Model. In order to represent documents as vectors we have to create the lexicon of words whose size will be the one of our vectors space and then a document term frequency matrix.
Two main steps have been followed:

- Tokenization
- Cleaning

Texts are tokenized and each token is passed to the cleaner before being added to the lexicon.
For the tokenization ***nltk library*** has been used. This library allows the tokenization by sentence and by word, so two FOR-loop has been used and the punctuation caught as individual token.
The cleaning consisted of four operations: eliminating numbers, eliminating stop-words, eliminating tokens with less than 3 characters and stemming.
The phase of stemming consists in keeping just the roots of words in order to consider different forms of the same verb or noun as the same feature. Since in the following analysis it is always used the array of stemmed tokens as documents feature it may be convenient to create a data object that relate the stems to the full tokens. Using ***pandas library*** a data frame has been created using stems as indexes (rows) and one column with the corresponding token (to obtain the two list we just need to include and exclude the stemming phase). This array turned to contain 5,618,845 terms: the so high number appears because terms are repeated (this way when searching for a stem just the first matching token is returned, this is a limit).

3.2 Cosine similarity and Tf-idf matrix

At this point we can compute the cosine similarity of each document compared to the others and the tf-idf matrix. These will be respectively used in the hierarchical clustering and in the kmeans clustering.
The ***sklearn library*** provides useful functions: it is possible to obtain the tf-idf matrix from the row texts using *TfidfVectorizer* and from the tf-idf matrix compute the cosine similarity for each document with respect to the others.

```
1.  from sklearn.feature_extraction.text import TfidfVectorizer
2.  tfidf_vectorizer = TfidfVectorizer(max_df=0.8, max_features=200000,
3.                                     min_df=0.2, stop_words='english',
4.                  use_idf=True, tokenizer=tokenize_and_stem, ngram_range=(1,2))
5.  tfidf_matrix = tfidf_vectorizer.fit_transform(b_list)
```

As shown in the above code snippet the *TfidfVectorizer* can takes input parameters that allow us to vary the features we want to include in the tf-idf matrix:
`max_df`, maximum within document frequency for a term to be included in the matrix.
`max_features`, maximum number of features of the matrix
`min_df`, minimum percentage of documents a term must be in to be included (activated with `use_idf=True`)

`stop_words,` vocabulary of stop-words to be filtered `tokenizer=tokenize_and_stem,` we can assign to tokenizer a python function that tokenize and clean the text `ngram_range,` in this case we consider uni-gram and bi-gram.
The tokenizer function as specified is defined with the following code snippet:

```
def tokenize_and_stem(text):
    tokens = [word for sent in nltk.sent_tokenize(text) for word in nltk.word_to
kenize(sent)]
    filtered_tokens = []
    stems = []
    for token in tokens:
        if re.search('[a-zA-Z]', token):
            if token not in stopwords:
                if len(token) > 2:
                    filtered_tokens.append(token)
    for t in filtered_tokens:
        stemmer.stem(t).encode("ascii", errors="ignore")
        if len(t) > 2:
            stems.append(t)
    return stems
```

Where the text variable is our list of books.
Given this parameters it is possible to vary them and regulate the dimension and therefore the sparsity of the tf-idf matrix.

Parameters	**Tf-idf matrix features**
Filtering tokens shorter than 4 char max_df: 0.8 min_df: 0.2 ngram_range= (1,3)	28,036
Filtering tokens shorter than 3 char max_df: 0.8 min_df: 0.2 ngram_range= (1,3)	30,965
Filtering tokens shorter than 3 char max_df: 0.7 min_df: 0.3 ngram_range= (1,2)	10,917

3.3 Hierarchical clustering

I the hierarchical clustering we use the distance matrix computed as:

$$Distance\_matrix = 1 - Cosine\_similarity\_matrix$$

The distance matrix is a squared matrix whose dimension correspond to the number of documents. Each value is comprehended between 0 and 1 depending on the distance between two documents (with zeroes onto the diagonal line).
The ***scipy library*** provides a ward function that calculates the linkage matrix given the distance matrix, and a dendrogram function that allows us to visualize the dendogram. The result of hierarchical clustering on our document corpus starting from the second tf-idf matrix is shown in *Figure 3*.

3.4 Kmeans clustering and multidimensional scaling

In the ***sklearn library*** we find functions to execute these two operations: kmeans algorithm works with the tf-idf matrix while mds takes the distance matrix
The kmeans algorithm requires us to set the initial number of clusters. From the hierarchical clustering we can conclude that there are 6 main clusters 5 of which are well defined while 1 is more heterogeneous according to the titles. Results of the kmeans clustering are shown in the following table where the 5 most close words to the centroid are shown. Documents ave been plotted in two dimension with kmeans cluster colours *Figure 4*.

The results in the scatter plot and in the table show two different iterations of kmeans clustering. The results are different as the algorithm is subjected to reach local minima.

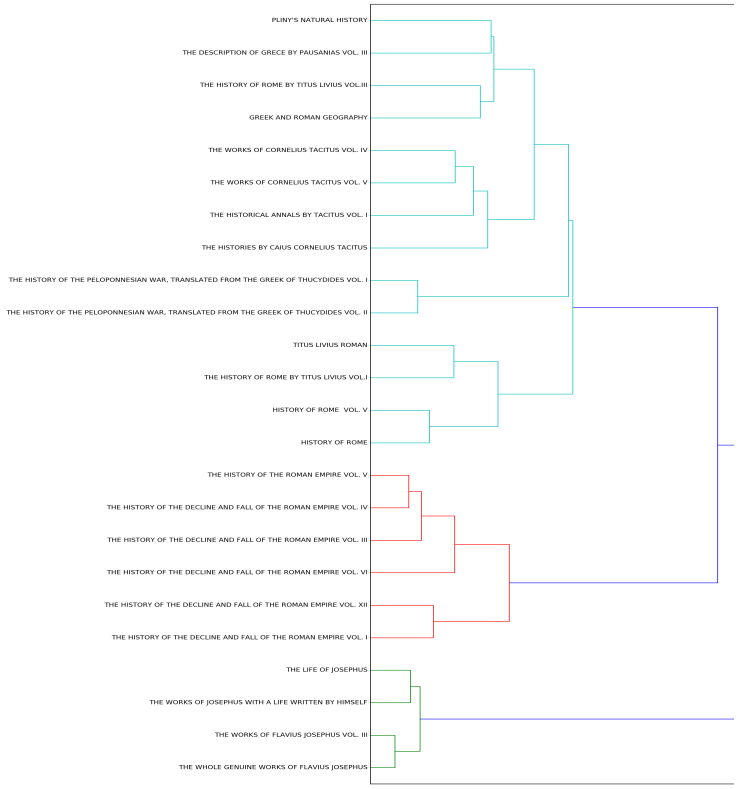

Figure 3: Dendrogram from hierarchical clustering

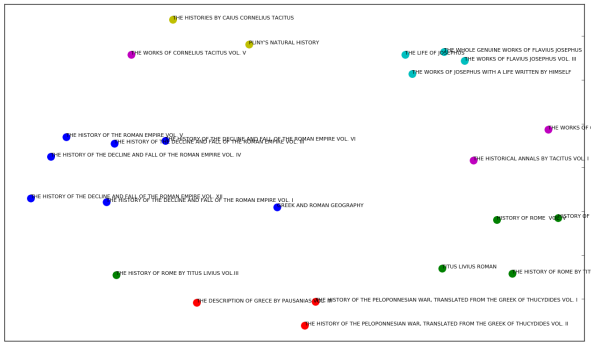

Figure 4: Documents plotted in 2D using MDS

(Blank page)