Big Data Mining - Assignment

Dimitrios Vogiatzis
CS2.18.0004

Victor Giannakouris - Salalidis
CS3.18.0002

1 Abstract

In this report we describe in detail the techniques that we used in the assignment for the Big Data Mining course. In summary, given an news input dataset the main goals of this assignment are the generation of a Wordcloud for each of the categories in the input dataset, article duplicate detection with respect to the cosine distance metric, the implementation and evaluation of several state-of-the-art classification algorithms, as well as a custom architecture that outperforms the aforementioned state-of-the-art algorithms in terms of a number of evaluation metrics. Through experimental evaluation we showcase that our architecture achieves better performance for all of the evaluation metrics that we have used, including accuracy, precision, recall, ROC and F-Measure.

2 Introduction

2.1 Goals

The goal of this assignment is the development of a system that will meet all the requirements by implementing modules for the following: 1. Generation of a Wordcloud for each of the available news categories, 2. Detection of articles with high degree of similarity between them (duplicate detection), 3. The implementation and evaluation of several state-of-the-art classification algorithms and 4. The development of a custom architecture that will outperform the algorithms defined in 3.

2.2 Installation

In this section we describe the steps required for installing and running our system. For these steps, an installation of *virtualenv*[1] is required.

1. Clone the repository:
   `git clone git@github.com:dcvogi/big-data-uoa.git`

[1] https://virtualenv.pypa.io/en/latest/

2. Create a virtual environment using *virtualenv*:
   `virtualenv .venv -p /usr/bin/python2.7`

3. Load the virtual environment:
   `source .venv/bin/active`

4. Install the requirements defined in *src/requirements.txt*:
   `pip install -r requirements.txt`

5. Run *main.py*:
   `python main.py`

After step 5 is completed, the results including *duplicatePairs.csv*, *Evaluation-Metric_10fold.csv* and all the .png Wordcloud images will be generated inside *results* folder.

3 Implementation

3.1 Wordcloud

In this task, we generated the Wordcloud for each of the input set categories. Wordcloud generation is a quite simple task. After a single pass to each document of each category the word frequencies for each category are generated. The Wordcloud of each frequency is generated with respect to these frequencies, were the size of a word image in the final word cloud results directly from its frequency. That is, the size of a word a will be larger that the size of a word b if a has a higher frequency. For this task we used the WordCloud[2] python package.

3.2 Duplicates Detection

In this section we describe in detail the methodology used in order to detect duplicate documents.

3.2.1 Vectorizer

In order to compute the similarity of the documents, we must first produce the term vectors for each document of the dataset (test set). To do that we used *TfidfVectorizer* from Python sklearn library. TfidfVectorizer transforms a collection of raw documents to a matrix of TF-IDF features. TF-IDF, short for term frequency-inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. [1] The TF-IDF formula is defined by the following equation[2][3]:

$$tfidf(i,j) = \frac{n_{i,j}}{|t \in d_j|} \cdot \log \frac{|D|}{|d \in D : t \in d|} \tag{1}$$

[2]https://amueller.github.io/word_cloud/

Where i the term index, j the document index, d a document, t a single term and D the whole document corpus.

3.2.2 Similarity detection

At this stage and after we produced the term vectors with TfidfVectorizer, we compute the cosine similarity between the vectors of each document versus the others, ending up with a $n \times n$ matrix where the value of the pair i, j corresponds to the cosine of the angle between. We consider this angle as the document similarity. In order to avoid double-checking the same pairs, we traverse the upper triangular of the matrix keeping only the pairs with similarity greater than the input threshold θ.

3.3 Classification Implementation

In this section we describe in detail the libraries that we used in order to implement the required classification algorithms, vectorizers, as well as dimensionality reduction modules.

3.3.1 Vectorizers

Two vectorizers were used for the assignment's purposes, that is, a bag-of-words (BOW) model and the Word2Vec (W2V) model.

Bag-of-Words. The bag-of-words[4] model is one of the simplest approaches used in text mining. A document is represented as a n-sized vector, where n the size of the dictionary. An element at the position i of a vector X represents the frequency of the i^{th} word of the dictionary in the vector X, where i denotes the word index. We used the *CountVectorizer* instance of scikit-learn library for leveraging the Bag-of-Words model. Below there is a sample code snippet.

```
from sklearn.feature_extraction.text import CountVectorizer

input_docs = open("docs.txt").readlines()
cv = CountVectorizer()
cv.fit(input_docs)
```

Word2Vec. Word2Vec[5] is a more complex model. In summary, Word2Vec is a two-layer neural network trained to reconstruct linguistic contexts of words, which are also called *word embeddings*[6]. In this model each word is represented as a vector of numbers, in contrast with conventional models like TF-IDF where each word is represented as a single weight number. The main benefit of representing words as vectors is that different words with the same meaning will be close each other in the vector space, i.e. the word "king" will be close to the word "queen".

To implement Word2Vec we used the gensim[3] library. As aforementioned, Word2Vec converts each word into a vector of numbers and thus, each input

[3] `https://radimrehurek.com/gensim/`

document vector is transformed into a vector of word vectors. This vector of vectors form leads into compatibility issues, as far as scikit's classification models take as input only vectors of numbers. To resolve that, we implemented a custom vectorizer class that overrides the three basic vectorization methods, that is, `fit()`, `transform()` and `fit_transform()`. The `fit()` method generates a Word2Vec model using the `from gensim.models import Word2Vec` class. The `transform()` takes as input a document in vector form. First, the document vector is being transformed into a vector of word vectors. Next, we generate an average vector, where each word vector is transformed into a mean value, resulting into a final document vector of means. Given an input vector X as follows:

$$X = \begin{bmatrix} w_1 \\ w_2 \\ . \\ . \\ w_n \end{bmatrix}$$

The $transform()$ method can be defined by equation 2 .

$$transform(X) = mean(X) = \begin{bmatrix} mean(w_1) \\ mean(w_2) \\ . \\ mean(w_n) \end{bmatrix} = \begin{bmatrix} w'_1 \\ w'_2 \\ . \\ . \\ w'_n \end{bmatrix} \tag{2}$$

Next, we use the produced mean vectors to feed our classification algorithms.

3.4 Neural Network Architecture

In order to achieve better performance than the algorithms described in subsection 3.3, we decided to proceed with changing both the classification algorithm and vectorizer used for experimentation, as well as performance optimization.

3.4.1 Vectorizer

To implement the proposed architecture we used Python's scikit-learn HashingVectorizer. HashingVectorizer applies a hashing function to term frequency counts in each document, making it an efficient way of mapping terms to features. Even though using a hash function can lead to collisions (e.g. distinct tokens mapped to the same feature index), with HashingVectorizer this is rarely an issue as there is a parameter, *n_features*, that can be tweaked to avoid collisions. The parameter *n_features* refers to the number of features (columns) in the output matrices. In our tests we experimented with different values for *n_features* and after a number of tests we concluded that with the given dataset, the most performing value was *n_features*=2^{18}. This is the recommended one for text classification problems.

3.4.2 Classification Algorithm

In the shake of experimentation we decided to implement a neural network for this task. The neural network used is a Multi-layer Perceptron classifier. A multilayer perceptron (MLP)[7] is a class of feedforward artificial neural network. A MLP consists of, at least, three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training. Multilayer perceptrons are sometimes colloquially referred to as "vanilla" neural networks, especially when they have a single hidden layer. A hypothetical example of a Multilayer Perceptron Network is depicted in Figure 1. In our architecture, the MLP consists of an Input Layer, a single Hidden Layer with 100 of neurons and an Output Layer. We experimented with the number of neurons and we concluded that the above architecture was sufficient to achieve the performance we needed.

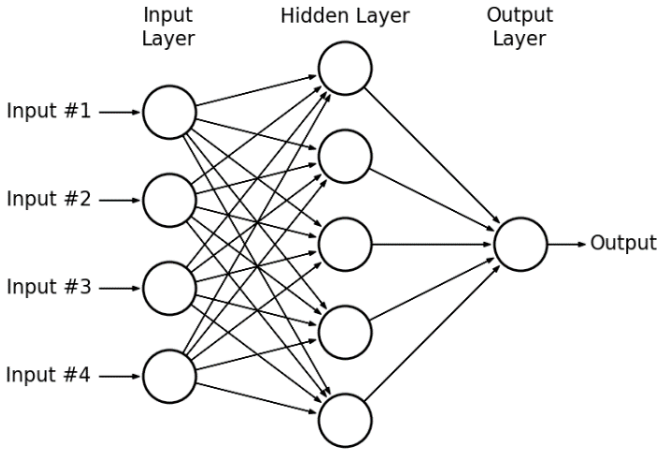

Figure 1: A hypothetical example of a MLP Network

3.5 Results

In this section we describe in detail the results from our experimental evaluation, which consists of the generation of wordclouds, document duplicate detection and classification algorithm performance comparison.

3.5.1 Wordcloud

For each of the input set categories we generated the following wordclouds:

Figure 2: Business

Figure 3: Film

Figure 4: Football

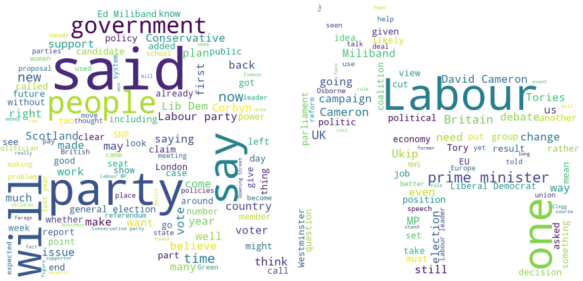

Figure 5: Politics

Figure 6: Technology

3.5.2 Duplicates Detection

As requested, the given source code accepts an input parameter θ as the similarity threshold. To export the results, we used $\theta = 0.7$ and based on this threshold, an output file (duplicatePairs.csv) is generated including all the document pairs with cosine similarity >= 0.7. Below is a small sample of the output file:

duplicatePairs.csv		
Document_ID1	Document_ID2	Similarity
10802	10213	0.7104
10802	13991	0.7073
6727	1015	0.7134
...	...	...
7811	11213	0.7150
14607	11213	0.7661

3.5.3 Classification Algorithms

Below you can find the evaluation metric table which holds the performance of each classification algorithm implemented:

Measure	SVM(BoW)	RF(BoW)	SVM(SVD)	RF(SVD)	SVM(W2V)	RF(W2V)	NN
Accuracy	0.935	0.927	0.931	0.929	0.935	0.889	0.964
Precision	0.932	0.925	0.928	0.927	0.930	0.885	0.962
Recall	0.929	0.916	0.925	0.920	0.929	0.878	0.961
F-Measure	0.930	0.920	0.926	0.923	0.929	0.881	0.962
AUC	0.956	0.949	0.954	0.951	0.956	0.925	0.976

Table 1: EvaluationMetric_10fold.csv

Testing the performance of the MLP, using the token vectors from *HashingVectorizer* we managed to achieve a performance increase over the methods used in subsection 3.3 as shown above, in the evaluation metric table (EvaluationMetric_10fold.csv). The performance of this architecture was consistently better as there were multiple experiments with different seeds and inputs. With our custom architecture (Neural Network) we generated an output file (testSet_categories.csv) which holds all the predicted categories for each document of the test set. The format of the output file is shown in the sample below:

testSet_categories.csv	
Test_Document_ID	Predicted_Category
10802	Football
523	Film
6727	Football
...	...
7811	Business
14607	Politics

References

[1] A. Rajaraman and J.D. Ullman. Data mining. In *Mining of Massive Datasets*, pages 1–17, 2011.

[2] Giannakouris-Salalidis Victor, Plerou Antonia, and Sioutas Spyros. Csmr: A scalable algorithm for text clustering with cosine similarity and mapreduce. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*, pages 211–220. Springer, 2014.

[3] Xenophon Evangelopoulos, Victor Giannakouris-Salalidis, Lazaros Iliadis, Christos Makris, Yannis Plegas, Antonia Plerou, and Spyros Sioutas. Evaluating information retrieval using document popularity: An implementation on mapreduce. *Engineering Applications of Artificial Intelligence*, 51:16–23, 2016.

[4] Bag-of-Words. https://en.wikipedia.org/wiki/Bag-of-words_model.

[5] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[6] Word embedding. https://en.wikipedia.org/wiki/Word_embedding.

[7] Multilayer perceptron. https://en.wikipedia.org/wiki/Multilayer_perceptron.