Advanced Big Data Analytics

Opinion Mining With Spark

Aurélien Pacard, Alain Soltani, Paul Hureaux

Under the supervision of
Michalis Vazirgiannis, LIX

Ecole Polytechnique
Route de Saclay, 91128 Palaiseau

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Context of study 1

2 A first framework : Preprocessing, Bag-Of-Words **2**
2.1 Preprocessing 2
2.2 Bag-Of-Words, TF-IDF 3

3 Improving our model : Graph-Of-Word & TW-IDF **4**
3.1 Graph-Of-Word 4

4 Going further : Doc2Vec & Paragraph Vectors **6**
4.1 Word2Vec : Learning Vector Representation of Words 6
4.2 Paragraph vectors 7
4.3 Predicting with Doc2Vec 8

5 Conclusion **10**

Bibliographie **11**

Chapter 1

Introduction

1.1 Motivation

Sentiment analysis, also known as opinion mining, is the analysis of feelings (i.e. attitudes, emotions and opinions) behind words using natural language processing tools. The new types of Internet content enforced new ways of data management which, as a consequence, caused new problems and opportunities to arise. Over the last decade a huge increase of interest in sentiment analysis research is clearly visible. The use of sentiment analysis to gain insights from unstructured text is becoming more widely leveraged as the years go by, as more and more industries are developing their data science facilities and aim at engaging more with their audience, their clients or their customers.

1.2 Context of study

In this project, we aim at tackling the problem of distinguishing between favorable and non-favorable movie reviews, on a very large set of data. We are given two sets of movie reviews:

1. A set of $25,000$ documents that contain labeled reviews either as positive or negative (50%-50%). This will be used for training.
2. Another set of $25,000$ documents containing unlabeled reviews that we need to assign labels to them. This set will be used for testing.

The reviews were taken from a review form that included a movie rating, in the range of 1 to 10. In this collection, up to 30 reviews were allowed for any movie, as to avoid an "overpopulation" of reviews with similar ratings. In the labeled dataset :

- Negative reviews were considered as such if they had a rating less than or equal to 4.
- Positive reviews were considered as such if they had a rating higher than or equal to 7.

Using this approach, "neutral" reviews are eliminated since they usually introduce noise to the dataset.

Various preprocessing, feature extracting and learning algorithms were examinated throughout this project. All scripts used throughout this project are available at the following GitHub repository : https://github.com/alsoltani/OpinionMining.

Chapter 2

A first framework : Preprocessing, Bag-Of-Words

Now we exposed the context and the data available for this challenge, let us present our first framework, providing a good basis for further work. Throughout this chapter, we will present the preprocessing made on the dataset, a feature extraction technique and a supervised learning scheme for the movie review classification.

2.1 Preprocessing

Our preprocessing routine involves five initial tasks : parsing the documents to remove HTML markup, replacing abbreviations in text via a home-made Regexp class, non-alphanumeric characters removal, lower-case conversion and finally splitting into individual words. The code for the code for the Regexp class as well as the replacement words are available in the GitHub repository.

```
# Remove HTML
review_text = BeautifulSoup(raw_review, "html.parser").get_text()

# Replace abbreviations
review_text = RegexpReplacer().replace(review_text)

# Remove non-alphanumeric characters
alphanumeric = re.sub(r"\W+", " ", review_text)

# Convert to lower case, split into individual words
words = alphanumeric.lower().split()
```

Alg.1. Permanent pre-processing steps.

Then, we optional steps were added : lemmatization, stopwords removal and finally, stemming. The choice of making these steps optional was driven by experience : the pre-processing phase is realtively tricky, as we observed the cleaner the documents, the lower the score. This was a general phenomenon seen throughout this process.
For instance, when integrating a stemmer, the score collapses, loosing almost 10% of efficiency ; this is quite understandable, as a stemmer might very well erase useful information within the original text. The following functions (a lemmatizer, and a stopword remover), gave us a lower score, for every framework we used, which we found quite paradoxical. A lemmatizer that transforms plurals in singulars, and a stopword removal that erases common words, reduce the

dimension of our problem with a dictionary size of 250 000 words, we approximately get to 70 000 words. Therefore, our model should have been more efficient with these two, because a lot of noise should have been erased through these steps ; unfortunately, this was not the case.

```
    # Lemmatize
    if lemmatize:
        wnl = WordNetLemmatizer()
        words = [wnl.lemmatize(w, 'n') for w in words]

    # Optionally remove stop words and/or stem (false by default, not used in Doc2Vec)
    if remove_stopwords:
        stops = set(stopwords.words("english"))
        words = [w for w in words if w not in stops.union(set(additional_stops))]

    if stem:
        stemmer = SnowballStemmer("english")
        words = [stemmer.stem(w) for w in words]

    # Return the processed document as a string.
    return " ".join(words)
```

Alg.2. Optional pre-processing steps.

2.2 Bag-Of-Words, TF-IDF

Our document representation used was the classical Bag-Of-Words model, with a TF-IDF weighting function. Recall that Bag-Of-Words represents each document as the multiset of its words, disregarding grammar and even word order but keeping multiplicity. The most commun term weighting function in both IR and text classification is TF-IDF :

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \times \text{idf}(t, D) \tag{2.1}$$

where $\text{tf}(t, d)$ is usually the raw frequency of a term t in a document d, $\text{idf}(t, D)$ the logarithmically-scaled fraction of the corpus of documents D that contain the word t :

$$\text{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}, \tag{2.2}$$

with $N = |D|$ the total number of documents in the corpus.

With a TF-IDF matrix of shape (25000, 75641) by training sklearn's TFIDFVectorizer after peforming our pre-processing routine, we held out 30% of the training set for validation, and fitted several supervised learning algorithms on the remaining set.

The best performance was obtained for the Logistic Regression with a $\mathbb{L}_2$ penalisation :
we obtain an accuracy score of 88.80%, and an AUC score of 0.956551. The cross-validation scores for a 10-fold scheme averaged at 86.47%, with a minimal accuracy of 85.36%.

Chapter 3

Improving our model : Graph-Of-Word & TW-IDF

The Graph-Of-Word model [1] is an improvement over the classical Bag-Of-Word model, which does not take into account word order and word dependence – a popular issue in natural language processing. Graphs have already been successfully used in search and information retrieval, and motivated Messrs. Rousseau and Vazirgiannis to introduce a graph-based document representation with associated graph term weights.

3.1 Graph-Of-Word

In this framework, each text document is represented as a graph-of-word that corresponds to an unweighted directed graph, its vertices representing unique terms, its edges the co-occurrences between terms within a fixed-size sliding window and whose edge direction represents term order. This model will link all co-occuring terms, without considering their meaning or function in the text.

```
    def populate_graph(word_list, dg, sliding_window):

        """
        For each position/word in the word list:
            add the (new) word in the graph
            for all words -forward- within the window size
                add new words as new nodes
                add edges among all word within the window.
        """

        for k, word in enumerate(word_list):
            if not dg.has_node(word):
                dg.add_node(word)

            temp_w = sliding_window
            if k + sliding_window > len(word_list):
                temp_w = len(word_list) - k

            for j in xrange(1, temp_w):
                next_word = word_list[k + j]
                dg.add_edge(word, next_word)
```

Alg.3. Function to populate a directed graph using a word list.

In this context, we can define a retrieval model as a function based on a term weight (TW) rather than restricting it to a term frequency (TF) : in a graph-based representation, it is simply the weight of the vertex corresponding to that particular term. Here, we choose the in-degree as the weight of each vertex, following experiments of [1]. Hence we obtain the following scoring function :

$$\text{TW-IDF}(t, d, D) = \text{degree}(\text{node}_t) \times \text{idf}(t, D). \tag{3.1}$$

After the preprocessing routine exposed in Chapter 2, the Graph-Of-Word model is used to create a (N, M) feature matrix on which we will evaluate our learning algorithms, N being the number of documents in the corpus, and M the number of unique words. The IDF values are either calculated for the training phase, or re-using the previous ones for the testing phase ; for each document, a graph is created, then populated as follows :

```
# Dictionary of centrality values.
centrality = nx.degree_centrality(dg)

...

# For all nodes
#   If they are in the desired features
#       compute the TWIDF score and put it in features[i,unique_words.index(g)].

for k, node_term in enumerate(dg.nodes()):
    if node_term in idf_col:
        features[i, unique_words.index(node_term)] = \
            centrality[node_term] * idf_col[node_term]
```

Alg.4. Extract of the code for feature matrix creation via the Graph-Of-Word model.

We then ported our code to Spark using pyspark. This very efficiently reduced the computing time of the overall algorithm, although the time spend building graphs remained longer than a classical TF-IDF procedure (about 3 minutes longer, on 70% of the training set).
Spark comes with several supervised learning algorithms via the MLlib library, such as SVMWithSGD, LogisticRegressionWithLBFGS or NaiveBayes. Let us compare the results obtained using these three algorithms, using our preprocessing routine and our PySpark implementation of the TW-IDF model ; in a similar fashion to the previous chapter, we held out 30% of the data to create a validation set. The best accuracy score reaches here 90.16%, a good improvement over the previous results, here again obtained with a Logistic Regression using a Limited-memory BFGS optimization algorithm.

Mllib model	Accuracy score
NaiveBayes	76.22%
SVMWithSGD	84.53%
LogisticRegressionWithLBFGS	90.16%

Tabular 1. Results obtained on the validation set using the Graph-Of-Word model & PySpark.

We also adjusted the parameters of our learning algorithm in regards with the accuracy score. For instance, we fitted the Logistic Regression model with a regParam equals to 0.003 and a sliding window equals to 3.
We also tried to reduce the dimensionality of our very large TW-IDF matrix by using a PCA. For unknown technical reasons, the PySpark implementation PCAMLlib does not work with an output matrix dimensionality greater than 66.000. We should have selected the features (words) depending on their tfidf. If it was too small do not consider it, otherwise put it as feature.

Chapter 4

Going further : Doc2Vec & Paragraph Vectors

For the same purpose of taking account of word ordering and semantics of the documents in our analysis, we introduce a model called *Paragraph Vectors* [2], an unsupervised algorithm that learns fixed-length feature representations from variable-length pieces of texts.
In this model, a vector representation is trained to be useful for predicting words in a paragraph. The paragraph vector is indeed concatenated with several word vectors from a paragraph and predict the following word in the given context. Both word vectors and paragraph vectors are trained by stochastic gradient descent and backpropagation. While paragraph vectors are unique among paragraphs, the word vectors are shared. At prediction time, the paragraph vectors are inferred by fixing the word vectors and training the new paragraph vector until convergence.

4.1 Word2Vec : Learning Vector Representation of Words

Let us start with a recent and popular model, succesfully implemented by Google for its well-known Word2Vec (available at code.google.com/p/word2vec/). Here, very word is mapped to a unique vector, represented by a column in a matrix W . The column is indexed by position of the word in the vocabulary. The concatenation or sum of the vectors is then used as features for prediction of the next word in a sentence.
More formally, given a sequence of training words $w_1, w_2, w_3, ..., w_T$, the objective of the word vector model is to maximize the average log-probability

$$\frac{1}{T}\sum_{t=k}^{T-k}\log p(w_t|w_{t-k},\ldots,w_{t+k}) \tag{4.1}$$

The prediction task is typically done via a multiclass classifier, such as softmax. There, we have

$$\log p(w_t|w_{t-k},\ldots,w_{t+k}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}} \tag{4.2}$$

Each y_i is unnormalized log-probablility for each output word i, computed as

$$y = b + Uh(w_{t-k},\ldots,w_{t+k};W), \tag{4.3}$$

where U, b are the softmax parameters. h is constructed by a concatenation or average of word vectors extracted from W.

After the training converges, words with similar meaning are mapped to a similar position in the vector space ; thus, one can obtain very interesting insights on similarities between words, supposedly close to each other.

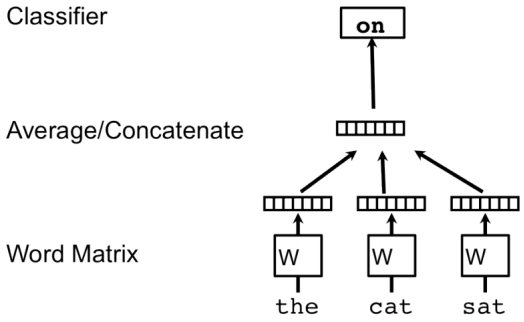

Fig.1. A framework for learning word vectors. Context of three words (”the,” ”cat,” and ”sat”) is used to predict the fourth word (”on”). The input words are mapped to columns of the matrix W to predict the output word.

4.2 Paragraph vectors

Inspired by the previous representation, we can develop a model for the paragraph vectors as follows. In the Paragraph Vector framework, every paragraph paragraph is mapped to a unique vector, represented by a column in matrix D and every word is also mapped to a unique vector, represented by a column in matrix W. The paragraph vector and word vectors are averaged or concatenated to predict the next word in a context.
The only change in this model compared to the word vector framework is in (4.1), where h is constructed from W and D. The paragraph token can be thought of as another word, and plays the role of a memory that remembers what is missing from the current context or the topic of the paragraph.

The paragraph vectors and word vectors are trained using stochastic gradient descent and the gradient is obtained via backpropagation. At every step of stochastic gradient descent, one can sample a fixed-length context from a random paragraph, compute the error gradient from the network in Fig.2 and use the gradient to update the parameters in our model. That additional randomness is very important for the efficiency of the training phase.

Suppose that there are N paragraphs in the corpus, M words in the vocabulary, and we want to learn paragraph vectors such that each paragraph is mapped to p dimensions and each word is mapped to q dimensions, then the model has the total of $N \times p + M \times q$ parameters (excluding the softmax parameters). Even though the number of parameters can be large when N is large, the updates during training are typically sparse, hence computable efficiently.

The paragraph vectors can be used as features for the paragraph, in lieu of bag-of-words. We can feed these features directly to our supervised machine learning algorithm, such as a logistic regression.

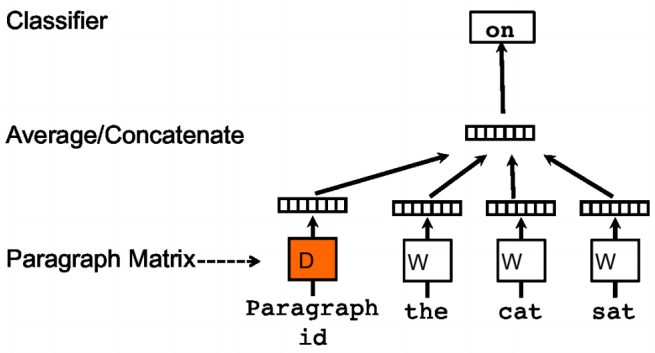

Fig.2. A framework for learning paragraph vector. This one is similar to the framework presented in Fig.1; the only change is the additional paragraph token that is mapped to a vector via matrix D. In this model, the concatenation or average of this vector with a context of three words is used to predict the fourth word. The paragraph vector represents the missing information from the current context and can act as a memory of the topic of the paragraph.

In summary, the algorithm itself has two key stages:

1. A training phase to get the word vectors W, softmax weights U, b, paragraph vectors D.
2. An inference phase, to obtain paragraph vectors D for new paragraphs (never seen before) by adding more columns in D and performing a gradient descent on D while holding W, U, b fixed.

4.3 Predicting with Doc2Vec

Following the previous supervised learning scheme, we fitted a Logistic Regression with a $\mathbb{L}_2$-penalty, onto the space panned by the features created via Doc2Vec.
The first implementation was heavily relying on the library gensim, which provides an efficient framework for Word2Vec and Doc2Vec.
Documents are initially fed into our model by using gensim's LabeledSentence class, simply taking a dictionary with keys as the file names and values the (unique) prefixes for documents from the corpus.
A new class LabeledLineSentence has been written to handle multiple documents at once. The constructor takes in a dictionary that defines the files to read and the label prefixes sentences from that document should take on.

```
class LabeledLineSentence(object):
    def __iter__(self):
        for source, prefix in self.sources.items():
            with utils.smart_open(source) as fin:
                for item_no, line in enumerate(fin):
                    yield LabeledSentence(utils.to_unicode(line).split(),
                    [prefix + '_%s' % item_no])
```

Alg.5 Extract of the code for our class LabeledLineSentence.

The parameters of the Doc2Vec model were set as follows :

- min_count : ignore all words with total frequency lower than this. This was set to 1, since the sentence labels only appear once.
- window: the maximum distance between the current and predicted word within a sentence. This was set to 3, in comparison with experiments of the previous chapters.
- size: dimensionality of the feature vectors in output. We first went for 100, considered a good number in this case, and then for 300, to try a more extreme setup.

However, when feature extraction performed by Doc2Vec did not had such a positive impact on the accuracy score as one might have expected. We only reached 86.53% of accuracy in the best case, with an output size of 300. The associated AUC score reached 0.937389.
Please note that due to the large number of documents processed, a pure Python calculation of the cross-validation scores (in a 10-fold scheme) could not be performed.

We then tried to port the Doc2Vec model in Spark, to accelerate computations. However, for unknown reasons, the training phase couldn't be computed in sufficient time (e.g. in less than dozen of hours) to provide evidence of the effectiveness of the parallelization. It is nevertheless a good focal area to dive in for further improvements, taking account of the recent progress made in other cases using this model.

Chapter 5

Conclusion

This project was a very insightful opportunity to get data science hands-on experience on a large dataset, covering a field of great interest that is semantic analysis. We could notably compare the performance of classical routine, such as Bag-Of-Words models and TF-IDF scoring functions, with state-of-the-art techniques such as Graph-Of-Word or Doc2Vec models.

In terms of performance, the best prediction was provided by the Graph-Of-Word model, with a best-in-class 90.16% accuracy. The graph creation in itself being a overall lengthy procedure, its partial implementation in Spark produced quick and efficient results, making this implementation a good alternative to the classical procedures.

Another area of development could be the adaptation of the Doc2Vec techniques, promising but untractable at the time, to the case of very large datasets, as the present one.

Once again, all the code used for this project is available on the following GitHub repository: github.com/alsoltani/OpinionMining. We welcome any opportunity and suggestion to improve the current results for further application.

References

[1] Rousseau, F. ; Vazirgiannis, M. (2013), *"Graph-of-word and TW-IDF: New Approach to Ad Hoc IR"*. In *CIKM '13 Proceedings of the 22nd ACM international conference on Information & Knowledge Management*
ACM New York. pp. 59-68, ISBN 978-1-4503-2263-8.

[2] Le, Q. ; Mikolov, T. (2014), *"Distributed Representations of Sentences and Documents"*. In *ICML '14 Proceedings of The 31st International Conference on Machine Learning*
ACM New York. pp. 1188-1196.