Machine Learning Engineer Nanodegree

Capstone project

Giacomo Sarchioni
March 6th, 2018

All the code used for the project is in the Jupyter notebooks available in the project's repository. Please use the .yaml files provided to build the appropriate conda environments.
A detailed mapping of the each paragraph to the corresponding Jupyter notebooks is reported in the Appendix.
The Appendix also contains a list of all the downloadable files that were not uploaded to GitHub due to size constraints.

I. Definition

Project Overview

My capstone project is about *sentiment analysis*, i.e. the field that "refers to the use of natural language processing, text analysis, computational linguistics, and biometrics to systematically identify, extract, quantify, and study affective states and subjective information"[1].
In particular, this project aims at classifying more than 500,000 Amazon Fine Food Reviews according to their sentiment (i.e. positive or negative), by only looking at the raw text. For example, if a product review says "What a fantastic product, I would highly recommend it!", then the algorithm should predict this text to be that of a **positive** review.

In general, traditional approaches to sentiment analysis rely on a count-based (absolute or relative) representation of text data. In this case, the degree of presence (or absence) of certain words - irrespectively of their ordering in the text - is the only determinant factor in explaining the sentiment. Recent developments in the field of deep learning, however, allowed for two significant changes:

- different text data representation by using word vectorisation (e.g. Tomas Mikolov's Word2vec[2]);
- new model architectures by using approaches that take into account the ordering of words (e.g. Recurrent Neural Networks or RNNs).

Such changes aim at providing a more comprehensive representation of both words and their ordered sequence, by potentially unlocking a semantic-like understanding of text. In other words, text data does not become a simple count-based representation of keywords, but an actual - yet numerical - representation of meanings and contexts.

Problem Statement

The goal of this project is to evaluate the performance of a deep learning-based classifier in predicting the sentiment (measured indirectly as product's rating) of a given review in the Amazon Fine Food Reviews' dataset available in Kaggle[3].
The goal is to verify whether a semantic-like representation of words and/or a complex neural network architecture do allow for improvement in the classification of text by sentiment.
I will use a common metric to compare my model against a plain-vanilla classifier that interprets text data by only looking at word counts, irrespective of the words' ordering and their meaning. In particular, I will train both models on the same set of Amazon's reviews and I will compare their performances on an un-seen, held-out dataset (the so-called test dataset).

---

[1] More details on sentiment analysis: https://en.wikipedia.org/wiki/Sentiment_analysis

[2] Original paper on Word2vec: https://arxiv.org/pdf/1301.3781.pdf

[3] More details on the Kaggle competition on Amazon Fine Food Reviews: https://www.kaggle.com/snap/amazon-fine-food-reviews and http://i.stanford.edu/~julian/pdfs/www13.pdf

Metrics

As I am going to show in Data Exploration paragraph, the dataset is **extremely unbalanced**. In particular, most of the reviews (more than 60%) are reviews about products that got a 5-star score.
The metric I chose, then, must be good at dealing with unbalanced datasets. *Area Under the Curve* (or AUC) is insensitive to such an issue, so I am going to choose it as the main evaluation metric for my project.

AUC represents the actual area under the Receiver Operating Characteristic curve (ROC curve). In statistics, such curve "illustrates the diagnostic ability of a binary classifier system as discrimination threshold is varied"[4]. The curve plots the True Positive Rate (TPR) of a classifier, against its False Positive Rate (FPR), for multiple discrimination thresholds. Such TPR-FPR combinations are used to plot a line (or a curve); the AUC metric is the area under such curve.
True Positive Rate is defined as:

$$TPR = \frac{TP}{P}$$

where TP is the number of true positives, and P is the number of positive samples (TPR actually corresponds to *recall*).
False Positive Rate is defined as:

$$FPR = \frac{FP}{N}$$

where FP is the number of false positives, and N is the number of negative samples.
It follows, quite intuitively, that TPR should be as high as possible (maximum value equal to 1), while FPR should be as low as possible (minimum value equal to 0).

The ROC curve is built by varying the discrimination threshold. For example, suppose that we have a binary classifier, trying to predict whether a certain input sample belongs to one category or another. Such prediction - let's call it *p* - can be thought as a float number between 0 and 1, representing the probability of that sample belonging to one class (while the probability of belonging to the other class would, naturally, be *1 - p*). The **discrimination threshold** is such value that tells at which level such prediction make the data sample belong to one class or the other. For example, if this threshold is 0.5, then if the predicted value *p* is greater than 0.5, the sample is assigned to one class, if below 0.5 to the other. By changing this threshold and by calculating TPR and FPR every time, it is possible to construct a Receiver Operating Characteristic Curve (ROC). The area under such curve is our metric AUC.

II. Analysis

Data Exploration

Dataset Size and Features

The original dataset of Amazon Fine Food Reviews contains 568,454 reviews. For each one of them, the following fields are reported:
- Id
- ProducId
- UserId
- ProfileName
- HelpfulnessNumerator
- HelpfulnessDenominator
- Score
- Time
- Summary

---

[4] More details on ROC: https://en.wikipedia.org/wiki/Receiver_operating_characteristic#Area_under_the_curve

- Text

Given the purpose of my project, *Score* and *Text* are the only important features I am really interested in. In particular:
- Text represents my input variable, i.e. the variable I am going to use to predict sentiment;
- Score represents my dependent variable, i.e. the variable I am going to predict.

A sample row of the reviews' data frame is reported below.

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenominator	Score	Time	Summary	Text
0	1	B001E4KFG0	A3SGXH7AUHU8GW	delmartian	1	1	5	1303862400	Good Quality Dog Food	I have bought several of the Vitality canned d...

In particular, the review above has a 5 score (i.e. maximum positive score) and the text is the following:

```
I have bought several of the Vitality canned dog food products and have
found them all to be of good quality. The product looks more like a stew
than a processed meat and it smells better. My Labrador is finicky and
she appreciates this product better than  most.
```

As you can see, the review's text is indeed positive. The purpose of my model would be able to read such input (i.e. the text) and predict the score associated to this review.

Duplicates

During my analysis I have noticed that there were multiple reviews that shared the exact same text. In particular, out of the 568,454 reviews, 174,785 reviews appeared at least two times in the dataset. For example, the review below appeared 7 times in the dataset.

```
Both" of Gloria Jean\'s "Hazelnut" and "Vanilla" packs of 50 contained
only "49."  Does Gloria Jean\'s have a fundamental packaging problem ??
```

This is a snapshot of the data frame of all the reviews exactly like the one above.

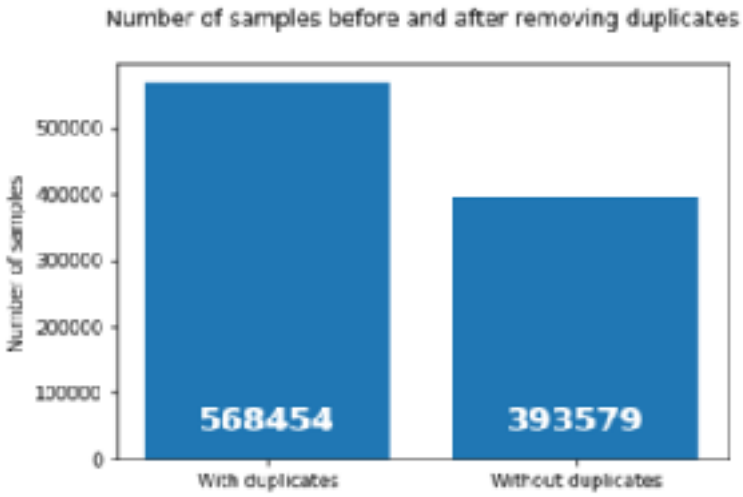
	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenominator	Score	Time	Summary	Text
164523	164524	B0082ZOIHY	A17950SQVNAVOD	Scott	0	0	1	1319625600	Packaging quality problem	"Both" of Gloria Jean's "Hazelnut" and "Vanil...
226879	226880	B0082ZOTUA	A17950SQVNAVOD	Scott	2	2	1	1319625600	Packaging quality problem	"Both" of Gloria Jean's "Hazelnut" and "Vanil...
353564	353565	B007TJGZ4A	A17950SQVNAVOD	Scott	2	2	1	1319625600	Packaging quality problem	"Both" of Gloria Jean's "Hazelnut" and "Vanil...
245224	245225	B0029KDZKI	A17950SQVNAVOD	Scott	2	2	1	1319625600	Packaging quality problem	"Both" of Gloria Jean's "Hazelnut" and "Vanil...
425081	425082	B003TQEWM2	A17950SQVNAVOD	Scott	2	2	1	1319625600	Packaging quality problem	"Both" of Gloria Jean's "Hazelnut" and "Vanil...
473106	473107	B000FIUK26	A17950SQVNAVOD	Scott	2	2	1	1319625600	Packaging quality problem	"Both" of Gloria Jean's "Hazelnut" and "Vanil...
516542	516543	B006N3IHHM	A17950SQVNAVOD	Scott	2	2	1	1319625600	Packaging quality problem	"Both" of Gloria Jean's "Hazelnut" and "Vanil...

It seems that UserId A17950SQVNAVOD, probably a guy named Scott reviewed a certain product with a very low score (1) saying that the main problem was a "Packaging quality problem". Why, then, do we have 7 exactly equal reviews by Scott?

The reason behind this seems that this dataset does not contain reviews, but **reactions to certain reviews**. Each of the 7 distinct lines for Scott's review are reactions from users who evaluated the helpfulness of such review.
In our context, however, we are not interested in predicting the usefulness of a review, but only its sentiment, indirectly measured by the score that Scott assigned to this specific product (i.e. 1).

Duplicates' Management Strategy
Do we need to remove such duplicated reviews or should we keep them? In theory, we are only interested in looking at text, and see how this text is associated with a certain review's score. In the case above, since the score value is actually the same for all the duplicated reviews, we are not introducing any inconsistency issue (i.e. same text, but different sample).
However, the existence of these duplicated reviews may introduce some problems when using certain methodologies that allow us to process text data. For example, in the case of **tf-idf vectorisation**, we count how many times a certain word appears in our review, but we also divide such count by the number of times the same words appear in the entire corpus. If we **artificially increase** the frequency of words simply because we have duplicated reviews, we are introducing a bias, i.e. we are going to make some words less important (because they appear more often in the corpus), while, in reality, they are not.
Therefore, my approach is to **remove all duplicated reviews**. This implies that I would check for text duplicates and just keep one of them.

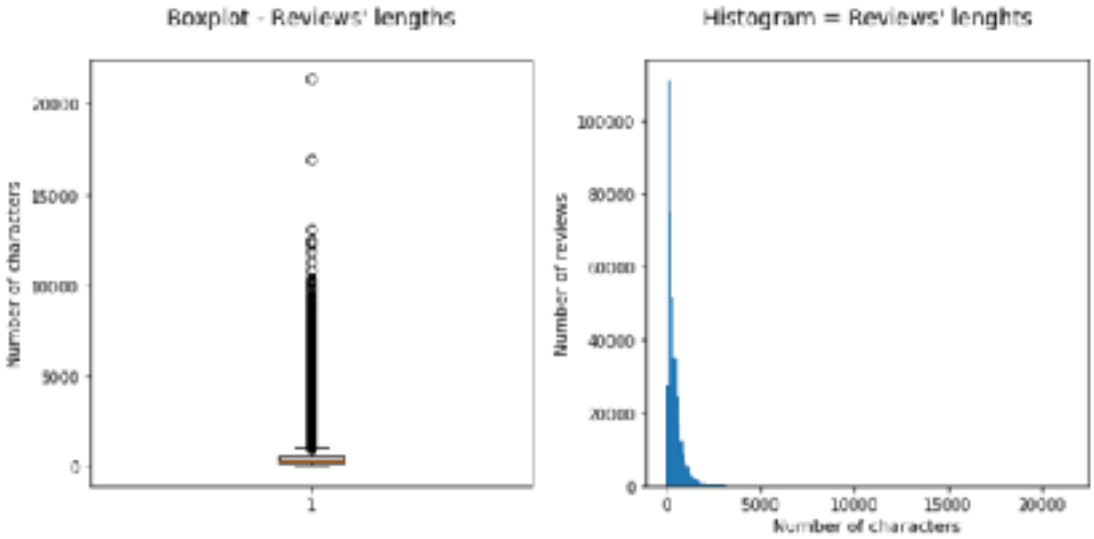
After removing duplicates, inevitably, my overall dataset became smaller. From 568,454 reviews I now have 393,695 reviews (as the chart below illustrates).



Exploratory Visualisation
Since, as said before, I will only be working with two features (i.e. Text and Score), I have performed exploratory visualisation only on these two.

Text
The first thing I looked at when analysing reviews is their length. To keep it simple, I simply use Python built-in `len` module which counts the **number of characters**. Here are some statistics:
- the **average** length of reviews is 434 characters;
- the **shortest** review is made of 12 characters;
- the **longest** review is made of 21409 characters.

As shown in the two charts below (a box plot and a histogram), the distribution of reviews' length contains a lot of outliers.
The skewed distributions of reviews' lengths, however, does not really affect my project. Clearly some users like to write longer reviews about the products they buy. I will show later if, possibly,



there is a difference between the length of a review and the score given to the product reviewed (maybe, when it comes to complaining about a certain product, users tend to write more - but this is just an assumption that I would need to validate).

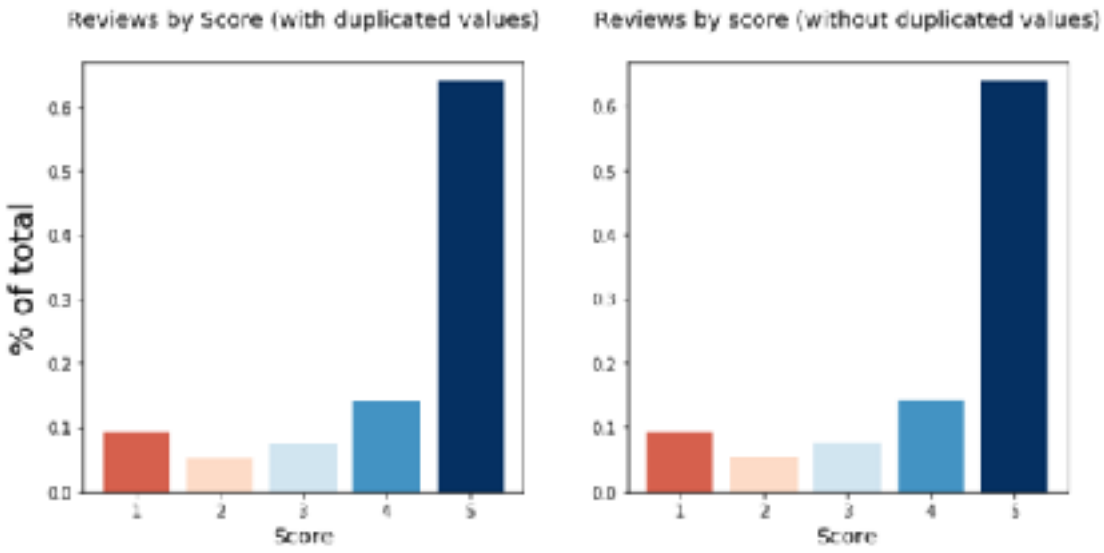
Score

Score is the dependent variable I am trying to predict. Possible scores assigned to a product are between 1 and 5, with 1 being the lowest and 5 the highest. The table below reports the percentage of reviews by score (after having removed duplicated reviews).

Score	Percentage of reviews
1	9.22%
2	5.28%
3	7.56%
4	14.24%
5	63.70%

As you can see, the majority of reviews are positive. This is something **very important** and it is the main reason why I chose AUC as my main evaluation metric. As explained before, AUC is a very robust metric when it comes to evaluating highly unbalanced datasets.

Another thing to check is whether having removed duplicates may have contributed to a change in the original distribution of scores. Let me show this with the two charts below.
As you can see, the two charts are pretty much the same. I am therefore quite confident in saying that having removed duplicated reviews did not change (or at least did not change significantly) the distribution of reviews' scores. This implies that the proportion of duplicated reviews must have been similar for all score classes.

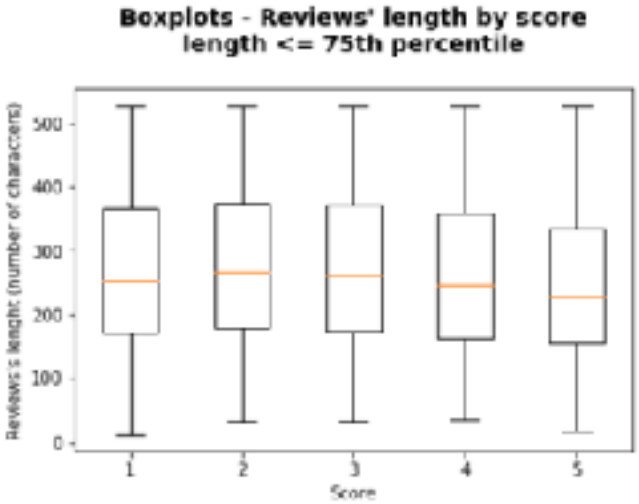


Reviews' length vs Score

Here I am trying to understand if there is a somehow evident relationship between the length of a review (measured as number of characters) and its associated score. If that is the case, then it may be a good choice to add `length` as an additional independent variable.

A good way to visualise such relationship is to use a box plot, where I show the distribution of reviews' lengths by score value.

Because of the outliers, it is difficult to see if there are any differences between reviews' lengths when looking at data by score value. Let me "zoom in" by looking at reviews whose length is smaller than the 75th percentile of all reviews' lengths.
It seems that reviews' length is pretty much similar when looking at the various score values. This implies that adding a feature like review's length to my set of independent variables should not add too much value. I will keep on using Text only.



III. Methodology

Data Preprocessing

When it comes to working with text data, a lot of data preprocessing is usually required. Let me cover this in the following paragraphs.

Text Parsing

Text Parsing - at least in my own vocabulary - refers to the activity of "formatting" text data so that it is as "clean" as possible. In particular, the parsing options I have envisioned in my project are[5]:

- removing any HTML text (e.g. HTML tags)
- make everything lower case
- removing non-letter characters (e.g. numbers, punctuations, etc.)
- removing stop words (i.e. very common words like "and", "or", "I", "about", etc.)
- stemming words (look here for more information).

I have then created a module called `review_parser.py` that allows me to - in a kind of functional programming fashion - apply a list of the parsing functions above to a given review.

It is important to have the **flexibility and modularity** to choose which parsing function/s to apply because for some of the activities in this project I would only need a partial text pre-processing. For example, when I will create word vectors I would actually need **not to remove stop words** since they are useful in the defining the "context" around a certain word.

Please see below the example of a review as originally written (left) and after parsing (HTML removal, lower case, non-letter characters removal and stop words removal).

Before text parsing	After text parsing
`These little guys are tasty and refreshing. I usually eat salads for lunch which can do terrible things to breath. I eat 2 of these after lunch (because they are so small) and I find this sufficiently resolves the problem. They're sugar free (sugar can actually help the smelly bacteria grow) which is great and they're tasty enough that I almost think of them as part of my lunch. This is a good bulk value buy and I'd highly recommend the mints themselves. Cheers!`	`little guys tasty refreshing usually eat salads lunch terrible things breath eat lunch small find sufficiently resolves problem sugar free sugar actually help smelly bacteria grow great tasty enough almost think part lunch good bulk value buy highly recommend mints cheers`

As you can see, HTML tags have been removed, all text is in lower case, there is no punctuation and stop words (e.g. "These", "I", "they", etc.) have been removed.

I have applied text parsing to create four sets of reviews with the following characteristics. Please notice that *Field* refers to the column names I have added to the reviews' dataframe.

---

[5] I have also referred to the parsing example provided in the Kaggle's tutorial on NLP *Bag of Words Meets Bags of Popcorn* (link here).

Set	Field	Parsing functions applied
0	*parser_zero*	• no HTML • lower case • no leading and trailing white spaces
1	*parser_one*	• no HTML • lower case • letter-only characters • no leading and trailing white spaces
2	*parser_two*	• no HTML • lower case • letter-only characters • no stop words • no leading and trailing white spaces
3	*parser_three*	• no HTML • lower case • letter-only characters • no stop words • stemmed words • no leading and trailing white spaces

Please notice that when I have applied the letter-only parsing function (i.e.. I have removed all non-letter characters), one review was left with no text. Such review was only made either of special characters or numbers. Since such text wouldn't have been really useful to understand sentiment, I have removed that sample.

Score Parsing

The objective of my project is that of building a model that can predict the sentiment of reviews. In reality, I am not really interested in whether the review is very negative (i.e. 1 score) or less negative (i.e. 2 score). Therefore, I am going to transform reviews' *Score* as follows:
• 1 and 2 *Score* reviews will be classified as 0;
• 3 *Score* reviews will be removed from the classification data;
• 4 and 5 *Score* reviews will be classified as 1.

Neutral reviews, while not used for the sentiment classification model, will be used as additional data for the word-vectorisation algorithm.

Summary
After having performed text parsing and score-adjustments, the dataset is now made of **393,578** reviews, distributed as below:
- 14.50% of the total reviews are negative
- 7.56% of the total reviews are neutral
- 77.94% of the total reviews are positive.

I have splitted the reviews into two *pandas* data frames and saved them as *pickle* files.

dataframe	description	file
neutral	Neutral reviews, i.e. *Score* equal to 3	`reviews/neutral_reviews/neutral.pkl`
sentiment	Neutral reviews, i.e. *Score* equal to [1,2,4,5]	`reviews/sentiment_reviews/sentiment.pkl`

Train, Validation and Test Split

An important part of my project is to split sentiment data into consistent train, validation and test sets. Only with a consistently-defined splitting I can truly evaluate the performance of various classification models.

In addition, as shown before, because data is unbalanced, I want my splitting to maintain the same proportion of negative-positive reviews across all the three sets.
To do that, I am going to use `StratifiedShuffleSplit` from `sklearn` (more information available here). Please notice that I firstly split the sentiment set into a non-test and test sets (90% and 10% of all sentiment reviews). The non-test set is then split again into a train and validation sets.

The table below reports the proportion of negative and positive reviews within the overall sentiment dataset, the train set, the validation set and the test set.

Set	Proportion of negative reviews	Proportion of positive reviews
sentiment	15.6853%	84.3147%
train	15.6853%	84.3147%
validation	15.6848%	84.3152%
test	15.6859%	84.3141%

Although with some minor differences, the proportion of negative-positive reviews across the various sets is maintained. Indexes are saved under `split_reviews/indexes.pkl` in the form of a Python dictionary.

In the next two paragraphs (*Bag-of-words* and *Word-vectors*), while not going into all the details, I will present an intuition for the BOW and Word2vec models. For more details, please refer to the indicated Jupyter notebooks.

Bag-of-words

As reported by Wikipedia (link here), in a Bag-of-word model (or BOW) "*[…] a text (such as a sentence or a document) is represented as the bag (multiset) of its words, disregarding grammar and even word order but keeping multiplicity*". In other words, such model operates as following:

- Create a word vocabulary, i.e. a token (an integer typically) is assigned to every word (or to the *n* most frequent words) that appear in a set of documents (or *corpus*);
- Every document is then represented as a vector whose size is the same as the word vocabulary defined where, for every index corresponding to a particular word, the model returns the number of occurrences that word appears in the document itself.

For example, suppose our word vocabulary is the following:

Index	Word
1	*i*
2	am
3	*udacious*

Therefore, the model will transform the following reviews as reported in the table below.

Original Text	Bag-of-word representation	Explanation
i am udacious	[1,1,1]	Each word in the vocabulary exactly appears one time
i am really udacious	[1,1,1]	Each word in the vocabulary exactly appears one time. Words not in the vocabulary are not counted
udacious udacious udacious	[0,0,3]	The word *udacious* appears three times
you love spaghetti	[0,0,0]	All the words in the original text are not indexed in the vocabulary

Let me comment on a couple of points.
First of all, "*i am udacious"* and "*i am really udacious"* are represented by the same BOW vector. This is because the word "*really*" is not in the word vocabulary. Clearly, however, the two sentences have slightly different meaning (with the second being more positive).
Secondly, if I were to change the order of words in the original text, my BOW vectorial representation would not change. BOW models do not retain any information about the ordering of words in a document.

Apart from the decision on whether to keep or remove stop words in my set of documents, there are a couple of options/parameters I will need to pay attention to. In particular, I am referring to:

- **word count methodology:** in the example above I have used a simple count of words in a document. A smarter - and typically more effective - way is to use the so-called tf-idf vectorisation. Such methodology takes into account the frequency of a word in the entire corpus, thus making very frequent word "less important";
- **vocabulary size:** this is a very important parameter. It is very difficult to have the right answer. Should the maximum size of my vocabulary be a certain percentage of the total number of unique words in my corpus? This is an important choice because having very long vectors may lead us into a problem of very high dimensional data, which, as a consequence, could bring us into the *curse of dimensionality's* trap. I have tested the performance of my classifier (in terms of AUC score) on the validation set for different vocabulary sizes and I chose the level that gave me the best result;
- **word definition:** as shown before, documents such as "*udacious*" and "*really udacious*" are very likely to be represented by the same vector. It is possible, **however**, to define words as "range of words". For example, "*really-udacious*" can be considered as a word itself. In `sklearn`'s vectorisers it is possible to do so by working on a parameter called ngram_range. I have noticed that setting this parameter such that ranges of **two words** are included in my vocabulary contributed to pick up items such as "highly-recommend", "really-like", "much-better", etc. etc. which, as you can see, do definitely carry some sentiment meaning.

Word vectors

In this project I have used a Word2vec model to represent my words in a vectorial way that carries semantic information. For a complete explanation of the model, originally developed by Tomas Mikolov, I strongly suggest to read this post by Chris McCormick.
The main decision I need to take here is whether to use a Skip-gram Word2vec model or a Continuous Bag of Words (or CBOW) model. While the end result is the same, i.e. a vectorial representation of words, the two models are technically different - if not opposite - and they may lead to different results.

Benchmark

As explained in the proposal document, I have defined two benchmarks for this project, i.e.:

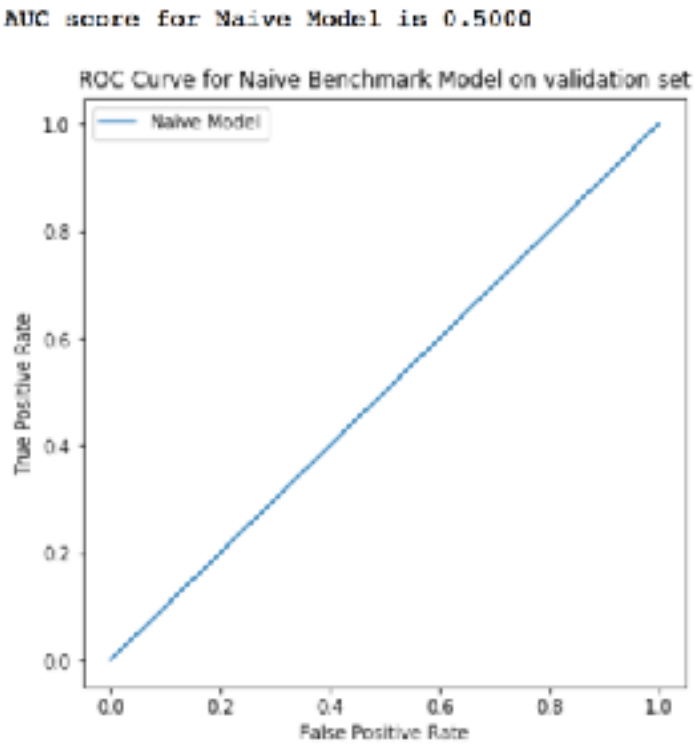
- a naive benchmark model, i.e. a model that always predicts a review to have the sentiment of the most frequent sentiment class. For example, if most of the reviews in the training set are positive, than the naive model will always say that a review is positive);

- a simple **Logistic Regression** model on tf-idf vectorised data.

Naive Benchmark

The naive benchmark model is very very trivial. However, I believe it perfectly acts as a reminder of the fact that we are in the context of of a very unbalanced dataset and an AUC score of 0.5 (i.e. that of the naive model) should really be our starting point.
If we were to measure the performance of the naive model in terms of accuracy, we would already get a model that is 84% accurate. That may make us think that the performance is good, but we know that, in reality, it is not.

The chart below reports the AUC score for the naive benchmark model on the **validation dataset**.



Logistic Regression on tf-idf data

The idea here is to use a very simple classification model like Logistic Regression on what is considered the basic approach when it comes to working with text data, i.e. bag of words.

The first choice I had to make here was to understand whether setting `ngram_range` equal to (1,2) was a better choice. As explained before, in this way I can in fact include in my vocabulary couples of words (e.g. “really-good”, “highly-recommend” etc.). To evaluate that, I did the following:

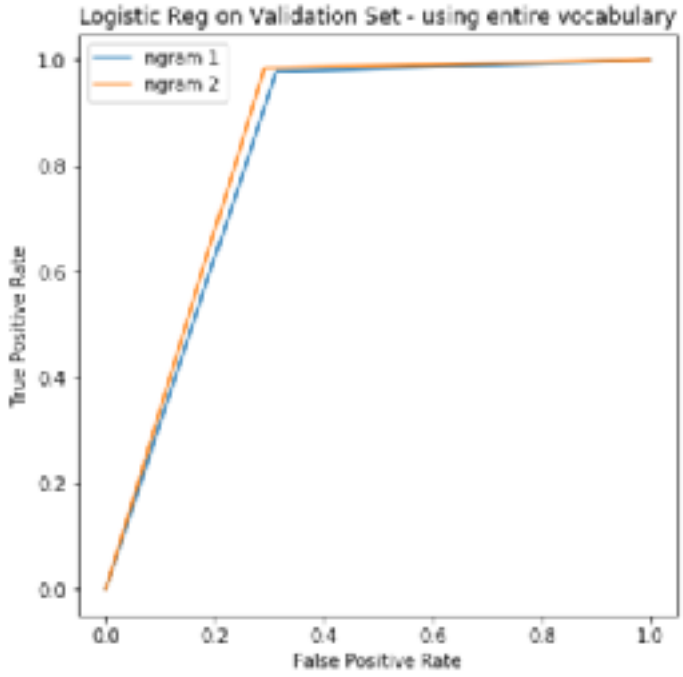
- Define two tf-idf vectorisers, one where `ngram_range` is the default value (1,1) and the other in which the `ngram_range` value is (1,2) - i.e. in the second case the model also looks at couple of words. I have used the default values for the other parameters;
- Fit both vectorisers on training data, and transform both training and validation data into tf-idf vectors;
- Train two default Logistic Regression model on the two different training datasets (i.e. one with `ngram_range` equal to (1,1) and the other with `ngram_range` equal to (1,2));
- Predict sentiment for the two validation sets;
- Calculate AUC scores and compare them.

The table below reports the performance on the validation set using two options for `ngram_range`. Please notice that I have used *parser_one* data (i.e. no HTML tags, lower case,

letter-only characters). I have included stop-words because they contain words like “not” which are actually very important when extending the `ngram_range` to 2.

ngram_range	AUC score on validation set
(1,1)	0.8324
(1,2)	0.8468

```
AUC score for ngram 1 is 0.8324
AUC score for ngram 2 is 0.8468
```

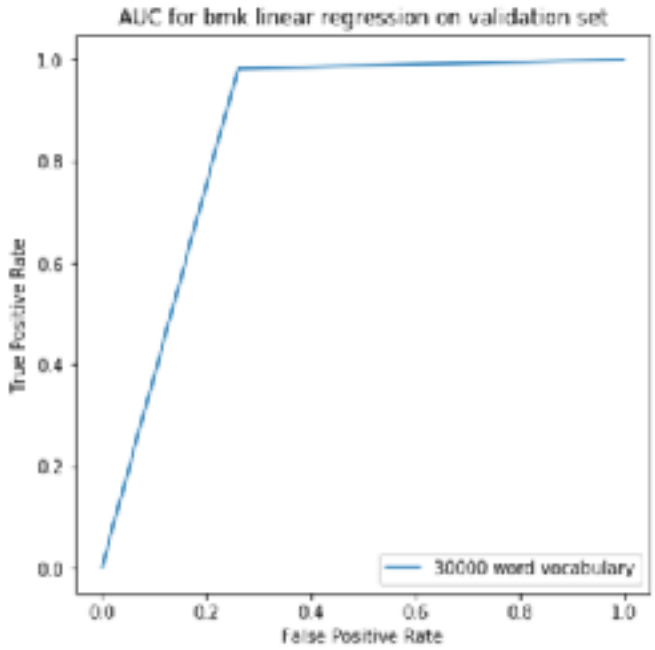


As you can see, the performance on validation set when using `ngram_range` equal to (1,2) is slightly better. Therefore, I am going to use this set-up.

Let me add a couple of points here. Just a simple logistic regression on tf-idf vectorised data led us to a very high AUC score. We are not looking at semantic meaning of words and their ordering, and, yet, we obtain a very good performance.

As a last step, I played a bit with the `max_features` parameter in the vectoriser. The model you have seen above, in fact, has been trained on the entire word vocabulary. In the case of `ngram_range` equal to (1,2), this vocabulary contains more than 2.5 millions of words or combination of words. This is way too much since it requires a lot of memory when fitting and transforming text data. In addition, we may well likely run into the *curse of dimensionality*’s trap. After having tested various options, I selected `max_features` equal to 30,000, i.e. the tf-idf vectoriser uses a word vocabulary with the most frequent 30,000 words (or couple of words). With this set up, I am using only 1% of the total vocabulary size and yet I am able to obtain an AUC score of **0.8609 on the validation set** (as shown below).

```
AUC score for 30000 word vocabulary is 0.8609
```



Let me show you some of the two-word items contained in the vectorisation vocabulary. They include:
- “the-worst”
- “worst-tasting”
- “ever-tasted”
- “so-bad”
- “never-buy”

As you can see, when taken together, the words above do carry a significant and/or stronger semantic and sentiment meaning than when taken individually. This is due to the fact of having set `ngram_range` equal to (1,2).

To sum, up the Logistic Regression benchmark model consists of a simple logistic regression fitted on data which is been transformed using a tf-idf vectoriser with `ngram_range` equal to (1,2) and `max_features` equal to 30,000. I have saved such model into an `sklearn`’s `Pipeline` object (the file is available in `bmk_models/bmk_log_reg.pkl`).

Algorithms and Techniques

The objective of my project is to evaluate whether the adoption and implementation of deep learning algorithms and techniques can improve the performance of my classifier, especially when attempting to retrieve “semantic” information from the reviews’ data.

As said at the beginning, I concentrated my efforts on two techniques:
- a new text data representation using a word vectorisation technique called Word2vec;
- new model architectures that are capable of taking into account the order of words.

According to whether I apply one or both the techniques simultaneously, I came up with a matrix of possible configurations.

	Without advanced DL architectures	With advanced DL architectures
Without Word2vec	Benchmark model	Conv1D with Keras embedding RNN with Keras embedding
With Word2vec	Logisitc Regression on averages of Word2vec's vectors	Conv1D with Word2vec embedding RNN with Word2vec embedding

---

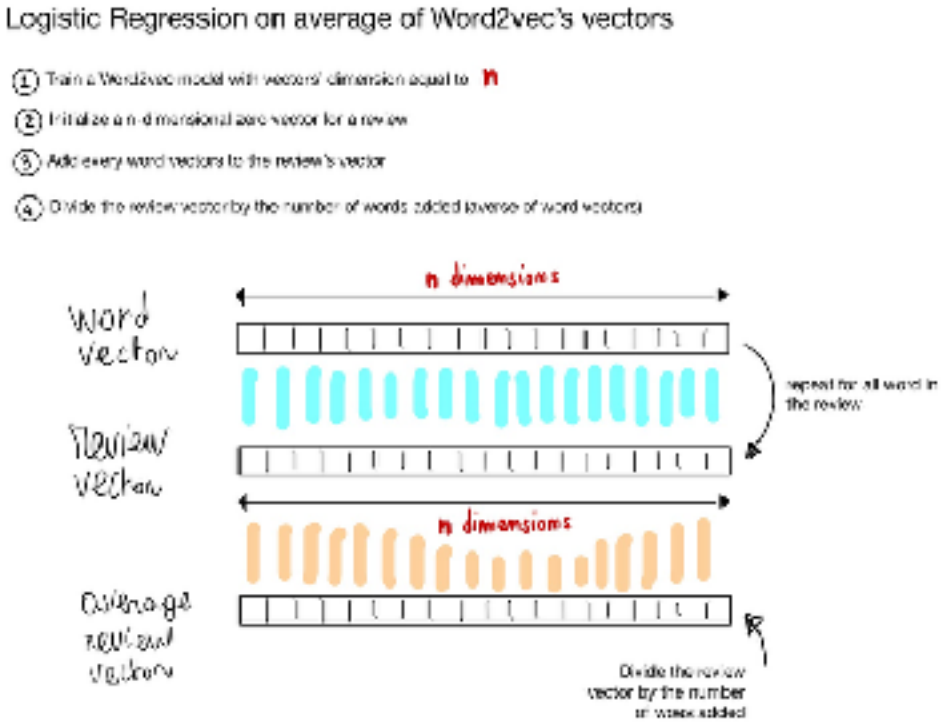
Benchmark model

Please refer to the paragraph on benchmark models presented above.

Logistic Regression on averages of Word2vec's vectors

In this option I am going to represent my text data using Word2vec. I find this algorithm to be extremely interesting.

I am going to train my Word2vec model on all sentiment reviews (excluding those kept for testing) as well as on the neutral reviews. The purpose here is in fact to have as much data as possible in order to learn an accurate representation of word vectors. I am using `gensim` library to train my Word2vec model.
Once I have my words defined as word vectors, I can "translate" a review into a sequence of word vectors. In particular, the review itself can be represented as a single vectors (with the same dimensionality of the word vectors) which is simply the average of all word vectors of the words within it. This approach is quite common and it has also been used in Kaggle's tutorial on NLP *Bag of Words Meets Bags of Popcorn* (link here).



Conv1D or RNN with Keras embedding

In this option I am evaluating the performance of two deep learning architectures.
Both of them require reviews to be transformed into an “embedding” matrix. The *Embedding* layer in the neural network will learn how to translate words into a word vectors.
While this may seem the exact same thing as the Word2vec model, there is actually a tiny, but significant difference. Here the representation of word vectors happen at the same time of the sentiment classification exercise. In other words, word vectors in the embedding layer are those that aim at minimising the loss on the sentiment classifier. Such word vectors, therefore, may not necessarily be semantic representations of words.

The output of the embedding layer is then passed to one of the two sequential architectures I have looked into, i.e. Conv1D or RNN.

Conv1D networks, as the name suggests, are **Convolutional Networks** on 1-dimensional data. Similarly to image classification, the aim here is to look at “windows” of words (represented by vectors) and identify “concepts” that can help us improve the performance of a sentiment classifier. If a dog classifier trained using convolutional networks was able to identify concepts like straight lines, curves, shapes, noses, eyes, etc., a convolutional network on text data should be able to find semantic relationships between words.

RNN, or **Recurrent Neural Networks**, while processing every input (i.e. a word vector) sequentially, are capable of retaining a “memory”, a “state”, of what came before and, through a series of “gates”, learn, during back propagation, which information to retain or discard. Here I will have to evaluate the performance of two very common RNN architectures, i.e. LSTM (or *Long Short-Term Memory*) and GRU (or *Gated Recurrent Unit*). At a high level, the two architectures are very similar in terms of concepts (i.e. information passes through a series of gates). GRUs do have, in layman’s terms, a simpler sequence of gates, thus typically making the GRU architecture more computationally efficient than LSTM.

Conv1D or RNN with Word2vec embedding

This option is exactly the same as the one above, with the exception that instead of training a word vector representation simultaneous to the sentiment classification exercise, I am going to use the vectorial representations learned in the Word2vec model.

A few words on Doc2vec

Instead of averaging the word vectors in a review (see above), I might use another technique called Doc2vec. This method is an extension of the Word2vec model. In addition to training a word vector representation, in fact, this model will train a vectorial representation for the document as well (i.e. a sentence).

First of all - and that applies to Word2vec too - I have to process my reviews by splitting them into sentences. A review, in fact, may consist of multiple sentences and it is important to avoid that words after a full stop (i.e. in another sentence) do not impact the vectorial representation of words (and “docs”) in the previous sentence.

As a consequence, even when using Doc2vec, I will have to transform each sentence in a review into its vectorial representation and then average out the doc vectors.

I have decided not to go ahead with the training of a Doc2vec model, but rather concentrate on the Deep Learning algorithms (Conv1D and RNN). I just wanted to mention, however, that using Doc2vec would be another option for this project.

Implementation

Training Word2vec model

This is probably one of the key steps in the *Implementation* section. I have used `gensim Word2vec` module in order to transform the words in my corpus of reviews into word vectors.

Reviews fed to the Word2vec model must first be split into sentences. This ensures that vectorial representations of words are only influenced by words around them in the **same sentence**. Splitting a paragraph into sentences is relatively using a punctuation tokeniser available in Python's `nltk` (more details available here).

The most important parameters to take into account when training a Word2vec model in `gensim` are[6]:

- **`sg`**, i.e. whether to use the CBOW (default) or skip-gram model;
- **`size`**, i.e. the dimensionality of the feature vectors. In Mikolov's original paper, size was set to 300. I will use the same value;
- **`seed`**, i.e. for the seed for the random number generator;
- **`sample`**, i.e. the threshold for configuring which higher-frequency words are randomly downsampled;
- **`negative`**, i.e. if > 0, negative sampling will be used. The integer value used in `negative` specifies how many "noise words" should be drawn (usually between 5-20). Default is 5. If set to 0, no negative sampling is used;
- **`window`**, i.e is the maximum distance between the current and predicted word within a sentence.

Let me further clarify some of these parameters below.

Word2vec's parameter - sg
A Word2vec model can be trained using two different architectures:
- skip-gram;
- CBOW, or continuous bag of words.

The difference between the two is in the input data and what we are trying to predict.

In the **skip-gram** architecture, the input is a single word (a one-hot encoded vector for the word), and the output is represented by the a vector that contains the probabilities that a randomly chosen word in the vocabulary is "nearby" the input word. In other words, the task of the model is that of identifying words that are likely to be around a given word (i.e. being its context).

In the **CBOW** architecture, on the contrary, the input to the model are the words around our target word (i.e. we feed the context) and we are trying to predict the most likely word that would fit that context.

I really like the dichotomy between the models as presented by Yanchuan Sim in a Quora's post (link here):
- in the skip-gram architecture, the task is that of **predicting the context given a word**;
- in the CBOW architecture, the task is that of **predicting the word given a context**.

Word2vec's parameter - sample

---

[6] https://radimrehurek.com/gensim/models/word2vec.html
Please notice that I have found an inconsistency in Word2vec API documentation. Default value for `sg` is 0. This means that CBOW would be the implemented architecture, and, when `sg` is 1, skip-gram architecture would be chosen. The documentation says exactly the opposite but I have checked in the original code in GitHub and my interpretation is actually right.

In word2vec models, it is possible to randomly downsample words according to their frequency. I will keep the default parameter (i.e. 1e-3, which represents the overall frequency threshold above which a word is considered **highly-frequent** and it is subject to downsample).

However, I would like to give a quick intuition about that. Consider the following three pairs of words:
- the dog
- the house
- the country

Assuming we are using a skip-gram architecture, the model will learn that the word “the” is very likely to be in the context of all the three words. This may lead to a result where the three words “dog”, “house" and “country" are represented by similar word vectors. In reality, however, we know that such words have a very different meaning. The fact that “the" is in their context, is purely a result given by the fact that “the” is an article and, well, it can be everywhere!
Through sub-sampling the model will randomly remove high-frequency words from the context and let the model:
- be more accurate from a semantic perspective;
- be faster, since we are passing fewer samples.

Word2vec’s parameters - negative
This is one of the coolest things in Word2vec. Typically, the size of a vocabulary will be very large. This means that we will likely have a lot of weights to train in the model and back-propagating the error can be a gigantic task.
Negative sampling solves this problem by letting the model update the weights for only a fraction of negative cases (i.e. the words that should not be in the context, if we are using a skip-gram model).

Conceptually this is similar to **batching** in the sense we are doing small updates a lot of times, rather than a big update a few times.

Results
After a few trials, I have noticed that using the skip-gram architecture leads to me better semantic representations. I will give you a quick example to show that. Below I am reporting the top five most similar words to the word “good” for both models, i.e. a CBOW model and a skip-gram model.

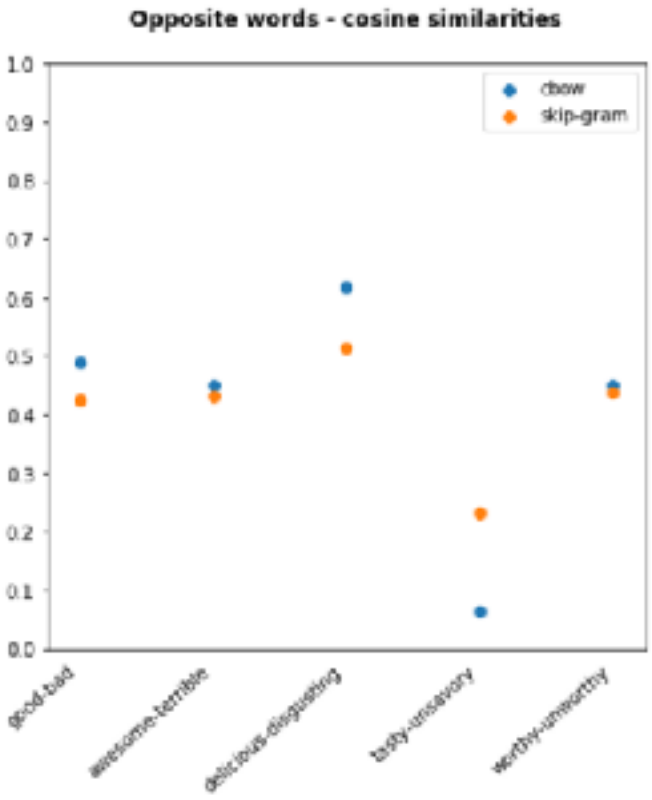
	CBOW Word2vec	**Skip-gram Word2vec**
Top 5 most similar words to “good”	great	goood
	decent	great
	fantastic	decent
	terrific	goood
	bad	agreat

Let me comment on this:
- the CBOW architecture said that “bad” is among the top 5 most similar words to “good”. This is clearly not ideal;
- the skip-gram model is capable of picking up possible typos (like “goood” instead of “good” and “agreat” instead of “a great”).

A trivial yet interesting way to evaluate performance of the two architectures is that of comparing the **cosine similarities** between couples of opposite words. In particular, I looked at the following opposites:
- good vs bad;
- awesome vs terrible;

- delicious vs disgusting;
- tasty vs unsavoury;
- worthy vs unworthy.

Because these words are opposite, we should have a low value for cosine similarities[7].
As you can see from the chart below, the skip-gram model over-performs (i.e. lower cosine similarity between opposite words) the CBOW model in most of the cases. For the "tasty-unsavoury" couple of opposite words, the CBOW model is actually better.



Overall, I have decided to prefer the skip-gram architecture. For all the other parameters, I kept the default values, while I have set the dimensionality of my word vectors equal to 300.

Fitting Logistic Regression model on averages of Word2vec's vectors

Now that I have my word vectors, I can train my first "enhanced" model.
As explained before, for every review, I will initialise a zero vector with the same dimensionality I used to represent my word vectors.
For every word in the review, then, I will add its vectorial representation to the review vector, while keeping a counter of the words progressively added.
Finally, I will divide the review vector by the number of word vectors added (this is equivalent to take an average of the word vectors).
I wrote a little Python utility for performing all the steps listed above (available as `average_word2vec.py`).

My new average vectors represent the input data I will feed into the same Logistic Regression architecture defined in the benchmark model. The intuition is that the vectorial representation of words and - through averaging - reviews should be able to better capture semantic nuances which are not measured by count-based models like BOW.

---

[7] Cosine similarity is a value normalised between 0 and 1. Two vectors have a cosine similarity of 1 when they are parallel, and have a cosine similarity of 0 when they are orthogonal.

Conv1D/RNN with Keras embedding

Text data fed to a Conv1D neural network or a Recurrent Neural Network (RNN) must be tokenised (i.e. an integer index is assigned to every word in the vocabulary). Please notice that any model in this section is trained using *parser_one* data. This is because keeping stop words is actually very important here.

In addition, because the input vectors fed into the model must have the same dimensionality, I have to transform reviews into vectors of equal size. This implies "clipping" my reviews by defining a parameter called `maxlen`, i.e. the maximum length of a review's document (where length is measured as number of words). In particular:

- reviews which are longer than `maxlen` would be clipped at `maxlen`;
- reviews which are shorter than `maxlen` would be zero-padded, i.e. missing words will be replaced by a 0 index.

Keras' `pad_sequences` model makes all the above very easy to implement.

Just a quick comment on the `maxlen` value I have chosen for my model. I have looked at the length (in terms of words) of all *parser_one* reviews. See relevant statistics below:

```
Statisics for parser_one reviews' length

count    363824.000000
mean         78.942821
std          75.780450
min           2.000000
25%          34.000000
50%          56.000000
75%          96.000000
max        2486.000000
```

Since the median reviews length is 56 words, I though of choosing a `maxlen` value of 60 words. After some attempts, however, I have noticed that setting `maxlen` value equal to 50 words led me to slightly better results.

Once I have all my reviews tokenised, but before starting to add my Conv1D or RNN layers, I need to add an **Embedding** layer. Similar to Word2vec, an Embedding layer will create a vectorial representation of the words defined in the tokeniser. Such representation will be trained as any other parameter during backpropagation. It is important to specify the dimensionality of my embedding vectors - this is a parameter similar to the dimension of word vectors used in Word2vec.

This tutorial by Jason Brownlee provides a complete yet simple explanation on how to use keras' embeddings for sentiment analysis (link here).

Conv1D/RNN with Word2vec embedding

In this case, all the steps and models are exactly the same as the ones defined above, with the exception that the embedding layer is created by using the Word2vec vectors. Such layer, as a consequence, is **not trained** during backpropagation.

To do that, I had to create an *embedding_matrix* to pass to the Embedding layer. You can find more details in the notebook.

Preliminary results

The table below reports the AUC score on the validation set for all the models trained. Please notice that at this stage I have used pretty plain vanilla configurations (e.g. few layers, no dropout, etc.). All these models are available in the directory `deep_learning_models`.

Model	Vocabulary size	Embeddings depth	# Parameters	Validation AUC
Naive benchmark		-	-	0.5
Log Reg benchmark	30,000	-	30,0001	0.8609
Conv1D	106,684	50	5,344,426	0.8528
LSTM	106,684	50	5,363,770	0.8485
GRU	106,684	50	5,356,410	0.8474
Conv1D Word2vec	106,684	300	22,226	0.8193
LSTM Word2vec	106,684	300	93,570	0.8566
GRU Word2vec	106,684	300	70,210	0.8755

The table above offers numerous insights.

First of all, while I have trained my benchmark logisitic regression on a 30,000-word vocabulary (including couples of words), all the other models have been trained on a tokenised set that includes all the 106,684 single words in *parser_one* data.

In the models using a trainable embedding layer (i.e. Conv1D, LSTM and GRU), I have set the depth of my word vectors to 50. On the contrary, in the models using pre-trained Word2vec word vectors, the depth of such vectors correspond to the one used while the training the Word2vec model (i.e. in this case this value corresponds to 300).
That is why models that learn their own word vector representations have many more parameters to train (more than 5 millions).

Overall, the model's performance on the validation set does not seem to different from that of the benchmark model. The GRU model trained on Word2vec does actually achieve the highest AUC on the validation set.

However, the deep learning models here used are very simple. In fact, I have:
- not used very deep architectures (since the deepest model has only two layers);
- not trained my model for a high number of epochs (all models have been trained for 5 epochs only);
- not employed a common strategy to prevent overfitting like dropout.

In the next paragraph I will present my final model which is the result of multiple trials and errors aiming at identifying an improved architecture and an enhanced set of parameters.

Refinement

I have spent quite a long time testing different configurations of my deep learning models. In general, the three main items I have played with are:
- word vectors representations;
- depth of the model;
- architecture of the model.

With regards to **word vectors representation**, I have tested three possible options, i.e.:
- Keras embedding;
- Word2vec embeddings using the model I have trained before;
- Google Word2vec pre-trained vectors[8].

For the last two cases, I tested whether to further train the vectorial representation or not. This is somewhat similar to **transfer learning**. In other words, word vector representation is part of the

[8] http://mccormickml.com/2016/04/12/googles-pretrained-word2vec-model-in-python/

forward/backpropagation loop in such a way that even my word vectors are further trained in order to reduce the loss of my sentiment classifier.

With regards to the **depth of the model**, I have tried various configurations in which I have added hidden layers. In the case of convolutional models, for example, I have added more convolutional layers, also increasing the number of filters (i.e. a higher value for the `n_units` parameter). In the case of RNNs, I have tested models with two stacks of recurrent cells (either LSTM or GRU).

With regards to the **architecture of the model**, I kept on testing with the paradigms illustrated above, i.e.a 1-D Convolutional Neural Network, a LSTM RNN and a GRU RNN.

In addition to that, I have also experimented with other parameters, in particular:
- the size of vocabulary used to tokenise my reviews;
- the implementation of techniques to prevent overfitting like dropout and checkpointing my model only when validation loss was improving;
- the activation functions
- the pooling layers in the convolutional model
- number of epochs

In general, I have seen that the following changes contributed to an improvement in the performance of my classifier (measured as AUC on the validation set):
- **continue training** the word vector embeddings led to a higher AUC on the validation set;
- **applying Average Pooling on the first convolutional layers instead of Max Pooling** led to a higher AUC on the validation set;
- **convolutional networks** were less prove to overfitting. While RNN models were in fact able to bring down the loss on the training set much more, they were underperforming on the validation set when compared to convolutional models.

Therefore, my final model for the sentiment classifier is a convolutional model with the following characteristics:
- the model receives tokenised data on the entire vocabulary of *parser_one* reviews;
- tokenised reviews have a fixed length of 50 words and are zero-padded if their original number of words is below 50;
- the model has the following architecture:
    1. A Keras embedding layer whose weights are initialised as those of my Word2vec skip-gram model. The weights are **trainable**, i.e. they will be updated during backpropagation;
    2. A 1-D Convolutional layer with 32 filters
    3. A Leak-Relu activation layer
    4. An Average Pooling layer
    5. A 1-D Convolutional layer with 64 filters
    6. A Leak-Relu activation layer
    7. An Average Pooling layer
    8. A 1-D Convolutional layer with 128 filters
    9. A Leak-Relu activation layer
    10. A Max Pooling layer
    11. A 1-D Convolutional layer with 256 filters
    12. A Leak-Relu activation layer
    13. A Max Pooling layer
    14. A Flatten layer
    15. A Dense layer with 1024 nodes (activation `tanh`)
    16. A Dropout layer (probability of dropout equal to 50%)
    17. A Dense layer with 128 nodes (activation `tanh`)
    18. A Dropout layer (probability of dropout equal to 50%)
    19. A Dense layer with 32 nodes (activation `tanh`)
    20. A Dense layer with 2 nodes (activation `softmax`)

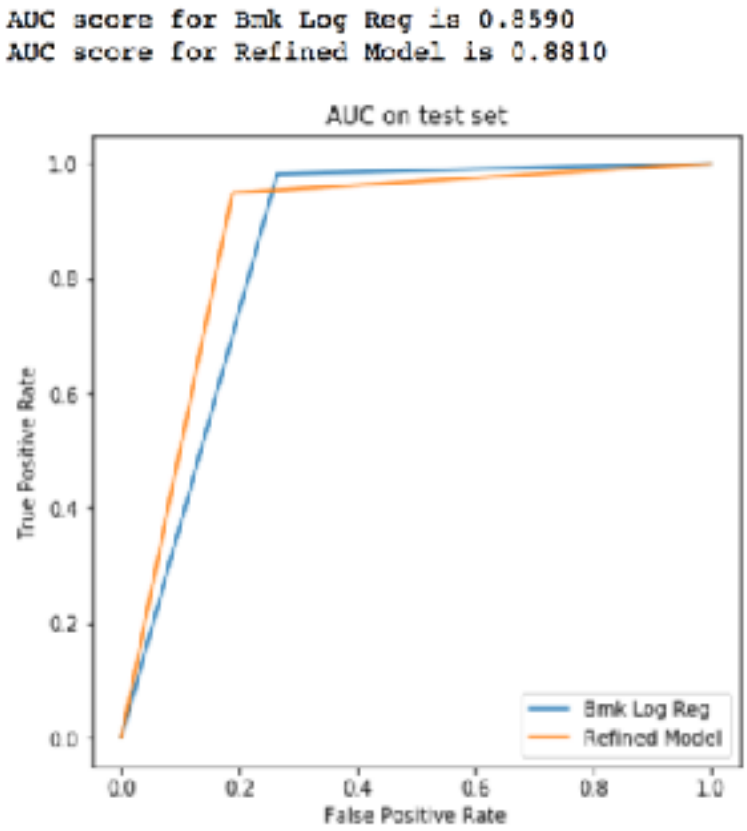
The model has been compiled using `rmsprop` aiming at minimising **binary cross-entropy loss**.

The performance of the refined model on the validation set against the benchmark model is reported below:

IV. Results

Model Evaluation and Validation

The chart below reports the performance of the benchmark models and the refined models on the test set.
As you can see, the deep learning model still performs better than the benchmark model.



The purpose of the project was to build a model that was capable of capturing semantic nuances. The benchmark model, based on bag-of-words data works pretty well, but should not be able to capture the importance of word ordering.
Let me show you this with an example. The text "I *bought this product and I found it not to be very good*" should be classified as a negative sentiment review. The probabilities for both models and classes are reported below.

Sentence: *I bought this product and I found it not to be very good*		
	Negative sentiment (probability)	Positive sentiment (probability)
Benchmark Model	0.0737	0.9263
Deep Learning Model	0.8618	0.1382

The table below reports the sentiment assigned to the first ten reviews in the test set (wrong predictions are highlighted in red).

Review	True sentiment	Benchmark model prediction	Deep Learning model prediction
bought this 1 lb merckens coco lite coatings to make chocolate covered strawberries for valentine's day. it tasted great, melted very well, dried quickly on the strawberries and it was great!	1	1	1
i didn't like this. i couldn't finish the entire bar and gave the rest to my husband who will for sure be getting these in his lunch until they're gone. he thought they were okay. they have a strange flavor i can't place. they're dry but it feels as though they added something to make it seem like they aren't. the banana and chocolate flavors aren't as noticeable as i thought they should be. i won't be buying these and i wouldn't recommend them.	0	0	0
i bought a package of these in amish country and had them with vegetable soup a couple nights ago. my husband, son and i all thought they were excellent. it made six so we all had two and wished we had bought more, so i saved the envelope in order to look them up was glad to find them here on amazon today. ordering more now. they are better than the frozen ones from schwann's but of course, not quite the same as red lobster's.all you do is add water and drop from a spoon - couldn't be easier. nice addition to any meal.	1	1	1
i love this coffee, but i am having trouble finding in the stores lately. this smells fantastic while brewing and is not overpowering to the taste. the coffee is rich and flavorful with just a hint of chocolate. only thing that would make this better would be if they sold it in k-cups.	1	1	1
the product was shipped fast and packaged in adequate shipping materials. i've never seen a bag of sour patch watermelon this big before, and i'm happy that i made this purchase.	1	1	1
nothing better than these pancakes with some coffee on a lazy sunday morning. best pancakes since grandma's.	1	1	1

Review	True sentiment	Benchmark model prediction	Deep Learning model prediction
i purchased these candy bars and yes they are great!!! just like the ones we had as children!!however, when i purchased them i didn't realize that they were being shipped by hometown favorites llc, the company that carries them, and the cost of shipping almost as much as the cost of the candy bar, $17.93.the charge was placed on my card and i was only aware of it when i got my notice that it had already shipped. i am hoping that if money is a consideration to someone that this will save them from my experience. amazon is great and upfront with their shipping costs -- i will not purchase anything from hometown favorites llc again.	0	1	0
this is really, really great tea. it's rich and delicious (especially with just a dab or milk or soy milk in it). makes me feel like i'm treating myself when i curl up with a cup of this awesome decaf tea. if you enjoy drinking different types of tea, then you'll love this aromatic, relaxing tea.	1	1	1
this gum's flavor is more akin to orange, or basically a sweet citrus taste. i chew a piece when i work out or play any type of sport, and it pastes at least 40 minutes to keep my mouth feeling refreshed. granted, the taste is obviously not as intense after the first 10 minutes or so. a few added bonus is that unlike some other gum like extra, the consistency of this gum doesn't degrade down to a nasty mush when your mouth's temperature gets very high. in addition, even when the fruity taste is mostly gone, there is no weird or bitter taste either.lastly, i'm not sure if vitamin b6 and b12 in this gum actually does anything, but for those who swear by these 2 vitamins for energy boost, they can't hurt.	1	0	1

Review	True sentiment	Benchmark model prediction	Deep Learning model prediction
i have been buying this product since i discovered it at costco, and recently discovered it on amazon. i find this to be a versatile and extremely useful ingredient in many foods i prepare ... casseroles, soups, stews, any non-sweet that requires flavor and liquid. it has the additional health benefits of chicken soup, and may be used in hot soup drinks for flu and colds. it makes an exceptional baked brown rice side dish when paired with himalayan pride basmati rice, which is also available here. i may sound like a salesman for this stuff, but i have made my own organic broths from vegetables and bones, and it takes a lot of time and care. for a prepared product, this has good flavor and versatility. and it is not canned, another large health plus - no leached metals, additives, or plastic liner contamination.i can't testify to the shipping practices of the various postal services. i have received damaged products in the past, and it was the responsibility of the seller to package them properly, and it is the responsibility of the shipper to handle the goods properly. usually, it seems to be the brutal treatment of the shipper in question, which means the seller must be notified. amazon stands by their products, so i buy a lot from them. when i have had a problem, they promptly took care of it. i find i am buying more from them every month, and will do so as long as they are honest with me, as i am with them.	1	1	1

It is worth looking into the two reviews that “fooled” the benchmark model.

The first one starts with “ *purchased these candy bars and yes they are great*” and that is probably enough to make our review be classified as a positive one. It seems common, however, for negative reviews to start with a positive sentence and then continue with the actual negative fact. The Deep Learning model was able to catch that and assign a negative label to such sentence.

The second one is an example in which a lot of negations are used. The user says something like “*this gum doesn't degrade down to a nasty mush*”. Our bag-of-word model picks up negative words like nasty, that without a context, may lead to assign such review to a negative label. The review ends with the sentence “*they can't hurt*”. Again, the verb “hurt” may be associated to a negative meaning but in this context such sentence is actually an expression of a positive (or at least non-negative) review.

Justification

The main differences between the benchmark model and my deep learning models are the word representation (i.e. bag of words vs word vectors) and the actual architecture (a simple logistic regression vs a convolutional network).

The first difference, i.e. the word vectors, allows for semantic-like representation of my words. In addition, input data is passed as a sequence of word, thus maintaining the ordering of words themselves.

The convolutional architecture acts as a pattern detector. Filters are applied to sequence of words aims at identifying higher-level concepts that influence the sentiment of a review.

However, while the AUC of the deep learning model is higher, I would like to test whether this a consistent result. Therefore, I am splitting my test reviews into a sub-samples of 100 reviews each, and calculate the AUC scores for both models for every one of them. I would then do a simple paired t-test to evaluate whether there is a true difference.

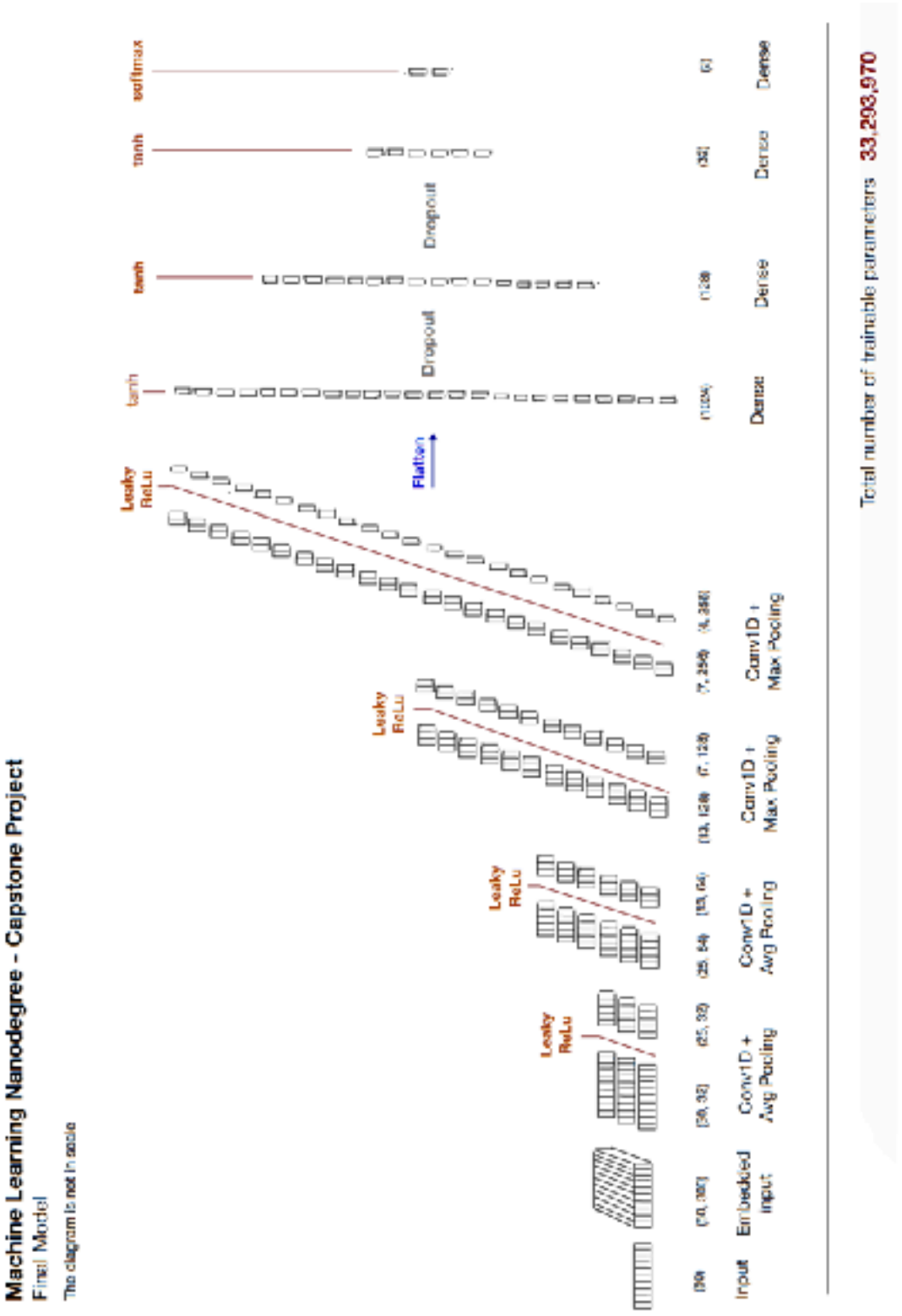
The p-value for the paired t-test is 1.1816195225222823e-11. Since such value is below 0.05 (a usual threshold set up for such test), I can safely reject the null hypothesis that assume both models to have the same AUC score. In other words, I am **happy to say** that my deep learning model is statistically more performing than the bow-based benchmark model.

V. Conclusion

Free-Form Visualisation

Throughout this report, I have presented a series of visualisations, covering exploratory data analysis, word vectors and performance evaluation.

The visualisation before is my handmade diagram that represents my final model. This is in line with a previous handmade diagram I made in the proposal submission document.



Reflection

The goal of this project is to evaluate the performance of a deep learning-based classifier in predicting the sentiment (measured indirectly as product's rating) of a given review in the Amazon Fine Food Reviews' dataset available in Kaggle.

During the project, I have done the following:
- Understand the nature of my dataset (exploratory data analysis);
- Format and parse text data;
- Process text data, either in a bag-of-words format or as word vectors;
- Identify a metric to evaluate the performance of any model;
- Fit a benchmark model;
- Design, fit and evaluate deep learning models.

If I had to choose two particular aspects of the project that I found interesting and/or difficult, I would choose the following:
- dealing with an unbalanced dataset;
- avoid the overfitting trap.

The first aspect is something that, in my professional life, I often found to overlooked. I work in finance and it is common to design and build model that aims at identifying relationships in the financial markets. Users, however, often tend to have a bias for what they are trying to predict (e.g. a stock market bubble), forgetting the relative frequency of such events in the historical context. That is why the choice of metrics that are not robust enough when dealing with unbalanced datasets can lead to serious problems. As demonstrated here, using a naive model always predicting a review to be positive would have lead to very good results in terms of overall accuracy, but we know that such a model is actually a very bad one.

The second aspect is something extremely evident in deep learning models. The excitement of seeing the loss decreasing on the training set may lead you to think that your model is performing very well. In reality, however, this may be just a sign that your model is "memorising" and not really learning. When I started training my RNN models (either LSTM or GRU), I saw the loss on the training set going much lower than that in convolutional models. However, when tested against the validation set, they were not performing any better (if not worse). Again, this a very important aspect. In my job, a lot of colleagues make the terrible mistake of evaluating the quality of a regression model by only looking at its performance on the training set. This would inevitable lead to situations of overfitting.

In addition to that, obviously, I found it very challenging to train a deep learning model on a non-GPU machine. Luckily, I was able to use an AWS instance for deep learning and that really helped me to try multiple combinations of parameters for my deep learning models.

Improvement

There are some aspects of the implementation that could be improved and potentially may lead to improved performance. Some of them were actually already tested.

First of all, instead of using my Word2vec embeddings (i.e. pre-trained word vectors from the Word2vec model), I could use pre-trained embeddings trained on larger corpus. In my project I have actually used Google Word2vec pre-trained model but that didn't necessarily achieve a higher performance. Some other pre-trained models, however, may be tested. *GloVe* from Stanford NLP group (link here) could be used too. In general, it would be very easy to implement such change. Once the pre-trained vectors are downloaded, an embedding matrix for every vector must be initialised in the corresponding Keras embedding layer. This is nothing different from what I have already did with my Word2vec embeddings.

Secondly - and I have tried this too - it could be worth investigating different optimisation strategy. In my case, however, `adam` and `rmsprop` led to similar results.

Thirdly, it would be interesting to investigate the performance of a hybrid model that includes both a RNN cell and a convolutional architecture. This means that instead of applying the convolutions

on the word vectors, they would be applied on the sequencers returned by an LSTM/GRU cell. This is really easy to implement. In Keras, he RNN cell should have its parameter `return_sequences` equal to `True` and after that we could attach our convolutional structure.

Appendix

Paragraph-to-notebook mapping

Paragraph	Notebook	Environment
I. Definition	-	-
II. Analysis	notebook_1.ipynb	environments/mlnd_env1.yml
III. Methodology - Data preprocessing	notebook_1.ipynb	environments/mlnd_env1.yml
III. Methodology - Benchmark	notebook_2.ipynb	environments/mlnd_env1.yml
III. Methodology - Implementation - Training Word2vec model	notebook_3.ipynb	environments/mlnd_env1.yml
III. Methodology - Implementation - Fitting Logistic Regression model on averages of Word2vec's vectors	notebook_4.ipynb	environments/mlnd_env1.yml
III. Methodology - Implementation - Conv1D/RNN with Keras embedding	notebook_5.ipynb	environments/mlnd_env2.yml
III. Methodology - Implementation - Conv1D/RNN with Word2vec embedding	notebook_5.ipynb	environments/mlnd_env2.yml
III. Methodology - Refinement	notebook_6.ipynb	environments/mlnd_env2.yml
IV. Results	notebook_7.ipynb	environments/mlnd_env1.yml
V. Conclusion	-	-

Downloadable files

File	Description	Link
Reviews	Contains original reviews, parsed sentiment reviews (all parsers) and neutral reviews	https://drive.google.com/drive/folders/11mvElVWJzBfeZyvRrcr1akr_BfhlWn9m?usp=sharing
Word2vec models	Include skip-gram and CBOW Word2vec models	https://drive.google.com/drive/folders/1ezOuoF5V6rR8vlEUUyQ7Gc6BWnNldmpE?usp=sharing
Deep Learning Models	Plain Vanilla Deep Learning Models (i.e. average of Word2vec vectors, Conv1D, LSTM and GRU)	https://drive.google.com/drive/folders/1ezOuoF5V6rR8vlEUUyQ7Gc6BWnNldmpE?usp=sharing
Final model	Include Keras tokeniser and final deep learning model	https://drive.google.com/drive/folders/1y6fOyOJcHcr0WvvZAc5RBnFTSBVUlVIN?usp=sharing