Members: Javier Sin & Nicolás Pelegrín

Movie Reviews Sentiment Analysis

`GitHub repository!!`

Introduction

After some brainstorming, when we decided to follow the idea of Sentiment Analysis in the context of Movie Reviews, we searched on kaggle for existing models to inspire ourselves. This way, we ended up deciding to use one of them and tweak it so that we improve the work already done using techniques viewed in class and some other techniques that we saw in our home city University from Spain. Here's a link to the kaggle model.

Problem definition

For this project we are going to build a model whose goal is to be able to distinguish between positive and negative reviews given a movie review in text format.

To do so, we are going to use **Sentiment Analysis**, a method for processing and analyzing text to be able to extract its emotional tone and determine whether the text, or in this case movie review, is positive or negative.

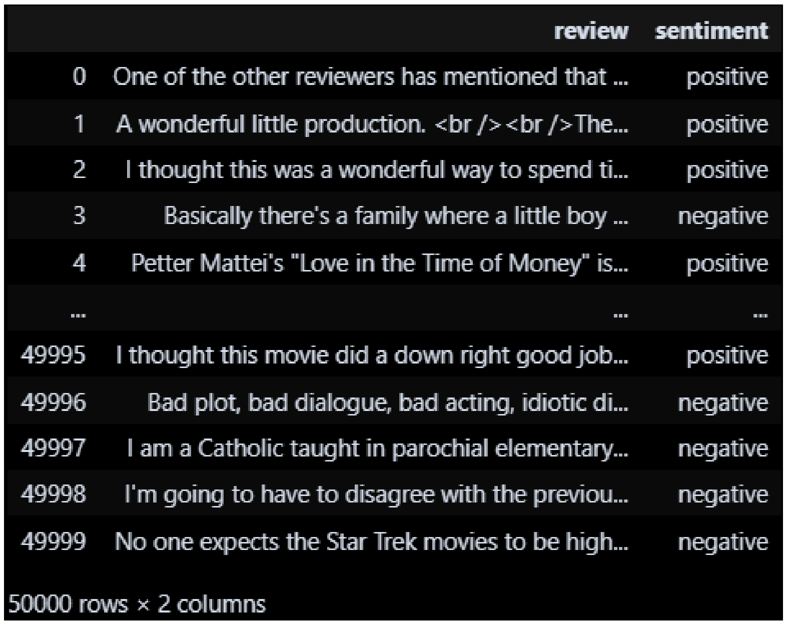
Together with this strategy we are going to try different model architectures to see which one gives the best results, at first Decision Tree, Naive Bayes and Logistic Regression.
After trying those *Supervised Learning Classification* models, we have decided to include the use of Neural Networks for this last part of the Final Project, along with the Recurrent Neural Networks.

Text preprocessing will also be an important part of this project, we will use **'Bag of Words'** (BOW) since we need to turn the movie reviews in text format into numerical vectors that our models can utilize and because with it we can account for the frequency of words and not their order which is not as relevant for our purposes.
Mention that RNNs will use the raw dataset without being vectorized. This will be explained later.

Data definition

The kaggle model comes with its respective dataset. It's based on a tensor consisting of 50000 rows by 2 columns. The features (columns) are the **review** itself, and the **sentiment** (positive || negative).

The dataset is based on a set of IMDB reviews. The *figure 1* shows a snippet of the data.



	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. <br /><br />The...	positive
2	I thought this was a wonderful way to spend ti...	positive
3	Basically there's a family where a little boy ...	negative
4	Petter Mattei's "Love in the Time of Money" is...	positive
...	...	...
49995	I thought this movie did a down right good job...	positive
49996	Bad plot, bad dialogue, bad acting, idiotic di...	negative
49997	I am a Catholic taught in parochial elementary...	negative
49998	I'm going to have to disagree with the previou...	negative
49999	No one expects the Star Trek movies to be high...	negative

50000 rows × 2 columns

***Figure 1*: Dataset snippet**

Mention that we have used a **train-test split** of **80-20** respectively. This way, from the 50000 reviews, 40000 are used in the context of training the models, and the remaining 10000 are left for the metric evaluation of the models performances.

Methods definition

In terms of the methods to be used, we're basing ourselves on **Sentiment Analysis** primarily, as mentioned before. To be exact, at first we used three different models: **Decision Trees**, **Naive Bayes** and **Logistic Regression**. In the former kaggle model, it was also used *SVM* but we haven't as we haven't seen it in class.

Here's a quick definition of the Naive Bayes classifier, as it hasn't been seen in class:
It's a group of supervised learning algorithms used for classification tasks. They are based on Bayes' theorem and assume that the presence (or absence) of a particular feature in a class is unrelated to the presence (or absence) of any other feature, hence the term "naive." This independence assumption simplifies the computation, making it computationally efficient.
In our case, we have verified that for example the Decision Tree model has taken way more time to train rather than the Naive Bayes one.

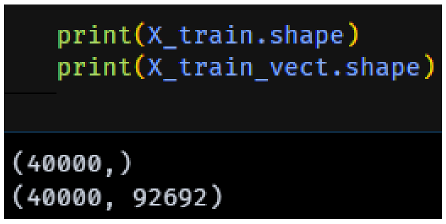
In addition, we have created a **Neural Network** and a **Recurrent Neural Network** and tuned it manually to try and improve the results obtained by the previous *Supervised Learning Classification* models.

Unlike the original kaggle model, where only the best default configuration model was tuned, we've created and tuned every one of the models from scratch so that we try to get the better performing model and we see the differences in the tuning for every one of them.

We'll also have to use methods for data preprocessing like **TF-IDF** (*Term Frequency-Inverse Document Frequency*), in order to **convert the text into numerical data**, this is the way in which we are going to represent the '*bag of words*'. TF-IDF is a statistical measure used to evaluate the importance of a word in a document relative to a collection of documents (corpus), which helps in extracting meaningful features from the text data.

- **TF** (Term Frequency): It's the first part of the algorithm, where the number of times a term appears in the document is calculated, to be then **normalized** by the total number of terms in that document.
- **IDF** (Inverse Document Frequency): After the Term Frequency, the results are measured to know how unique or rare a term is across all documents. This diminishes the weight of terms that are common across many documents, emphasizing terms that are more unique to specific documents.

As just mentioned, TF-IDF **normalizes the word counts by the frequency of the word** in the entire corpus, which helps in reducing the impact of common words that are less informative.
By converting text data into TF-IDF features, the model can better understand the importance of different words and phrases, leading to improved performance in tasks like classification.
Here's an example of the size for both regular training data, and vectorized training data:



```
print(X_train.shape)
print(X_train_vect.shape)

(40000,)
(40000, 92692)
```

As discussed in the notebook, the number of columns of the vectorized form (92692), makes allusion to the number of unique words achieved after the Tokenization, Stop words removal and other normalization steps such as lowercasing, punctuation removal, etc.

Mention that we're going to store each individual result for the performance of every model in a Panda's Dataframe so that we can compare all the models in the end of the notebook.

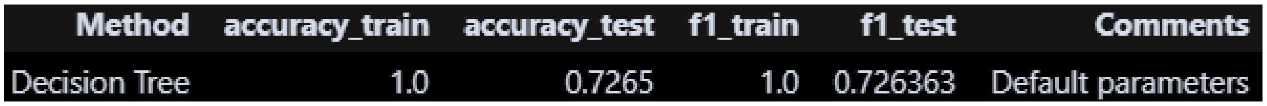
Experiments and analysis

Now we'll provide the achieved results for all the models. Even though we are gathering all the plots and values from the tests, **`we strongly recommend reading the notebook as there are more insights about the interpretation of the results`**.

After testing and tuning all the different models, we've achieved the following results for both **accuracy** and **f1_score**:

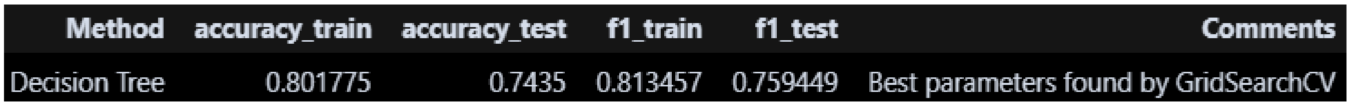
Decision Tree:

Default model:



Method	accuracy_train	accuracy_test	f1_train	f1_test	Comments
Decision Tree	1.0	0.7265	1.0	0.726363	Default parameters

Tuned model:



Method	accuracy_train	accuracy_test	f1_train	f1_test	Comments
Decision Tree	0.801775	0.7435	0.813457	0.759449	Best parameters found by GridSearchCV

The best parameters for the model are:

- 'max_depth': 20
- 'min_samples_leaf': 10
- 'min_samples_split': 2

As we can see, the default model suffers from **overfitting**, reaching the maximum accuracy. This means that the model has learned the patterns from the training data and struggles to generalize so the model has to be tuned to prevent it.

The *figure 2* represents the evolution of the model.

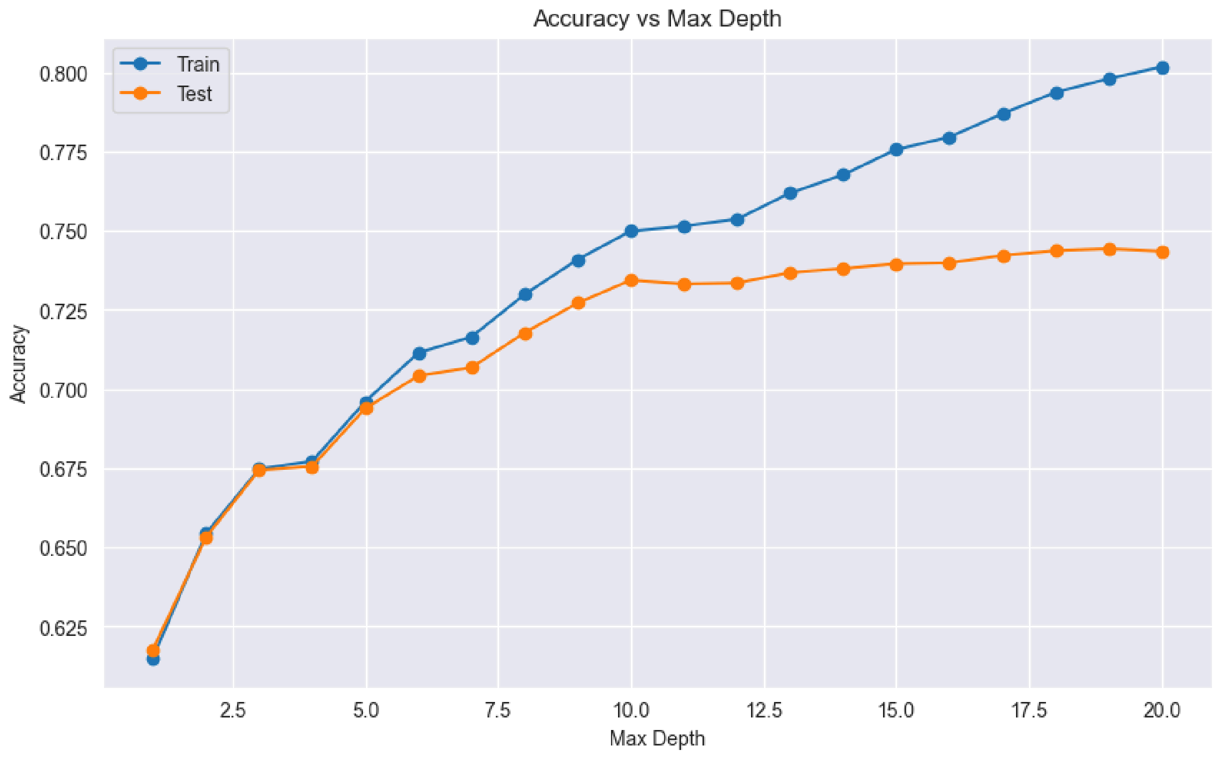
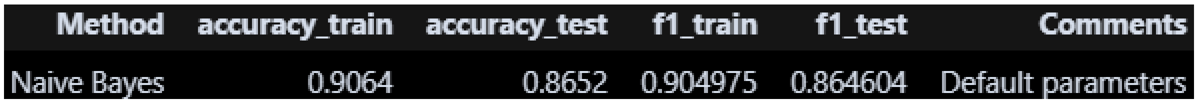


***Figure 2**: Decision Tree Accuracy vs Max Depth evolution*

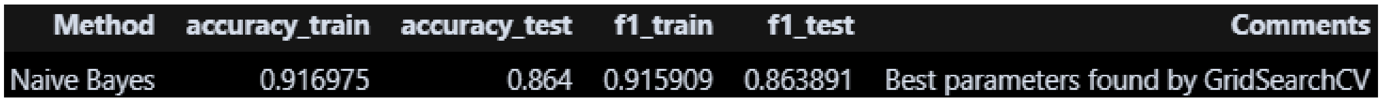
Naive Bayes:

Default model:



Method	accuracy_train	accuracy_test	f1_train	f1_test	Comments
Naive Bayes	0.9064	0.8652	0.904975	0.864604	Default parameters

Tuned model:



Method	accuracy_train	accuracy_test	f1_train	f1_test	Comments
Naive Bayes	0.916975	0.864	0.915909	0.863891	Best parameters found by GridSearchCV

The best parameter for the model is: 'alpha': 0.4789

If we compare the model without and with tuning, it's visible that even after tuning it, the base one is better in terms of new unseen data as the test metrics are better (by little).
The *figure 3* represents the evolution of the model.

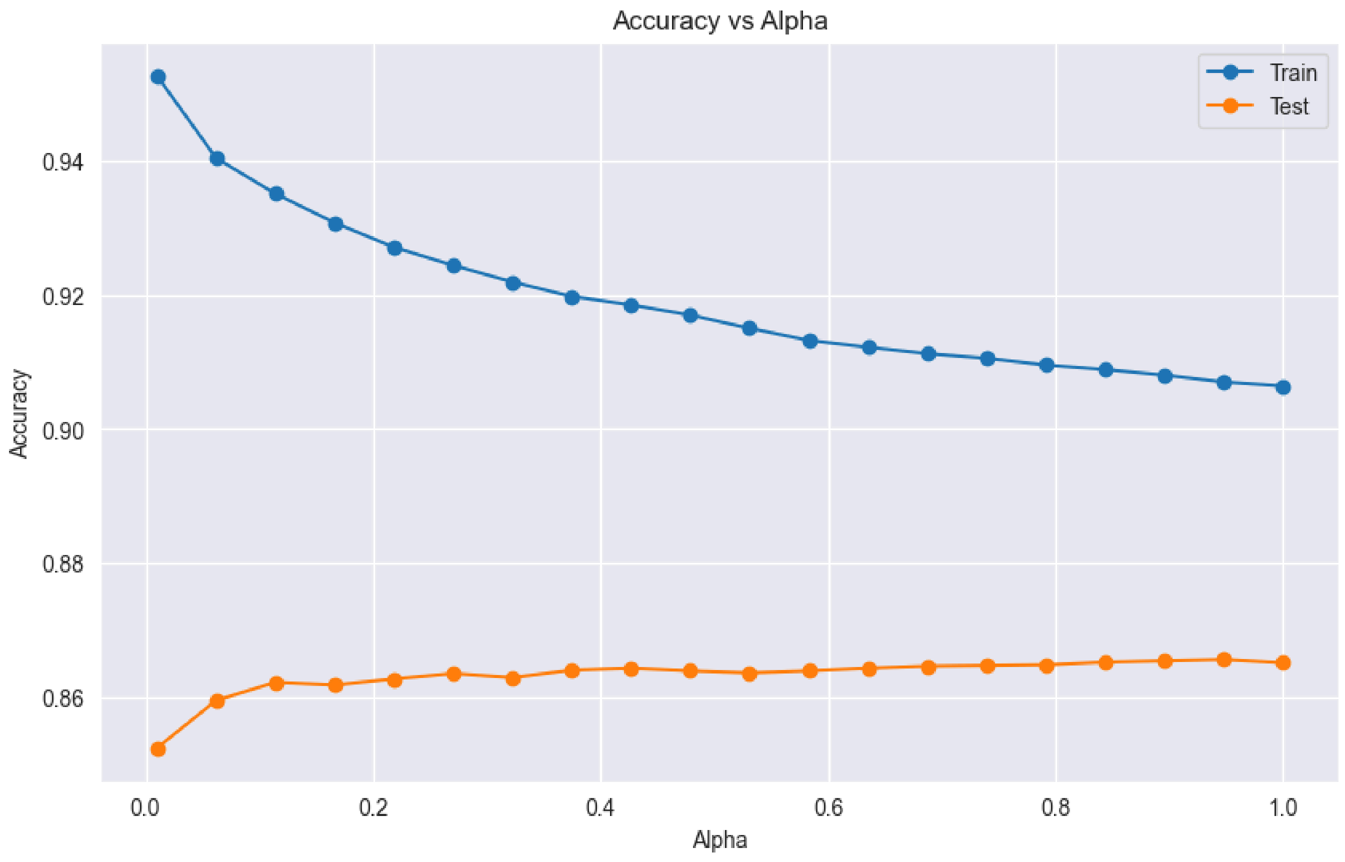
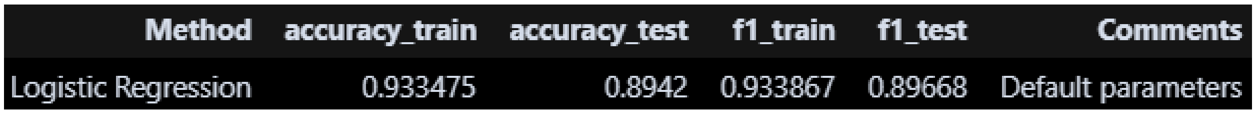


***Figure 3*: Naive Bayes Accuracy vs Alpha evolution**

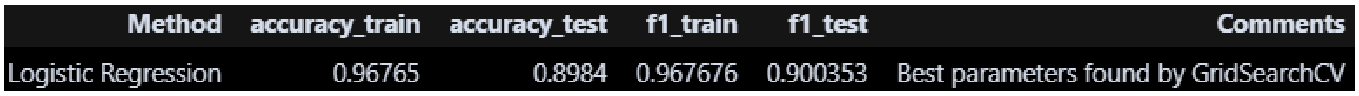
Logistic Regression:

Default model:



Method	accuracy_train	accuracy_test	f1_train	f1_test	Comments
Logistic Regression	0.933475	0.8942	0.933867	0.89668	Default parameters

Tuned model:



Method	accuracy_train	accuracy_test	f1_train	f1_test	Comments
Logistic Regression	0.96765	0.8984	0.967676	0.900353	Best parameters found by GridSearchCV

The best parameter for the model is: 'C': 4.2813

In this case, we can see that the tuned model improves both in terms of train and test data.
The *figure 4* represents the evolution of the model.

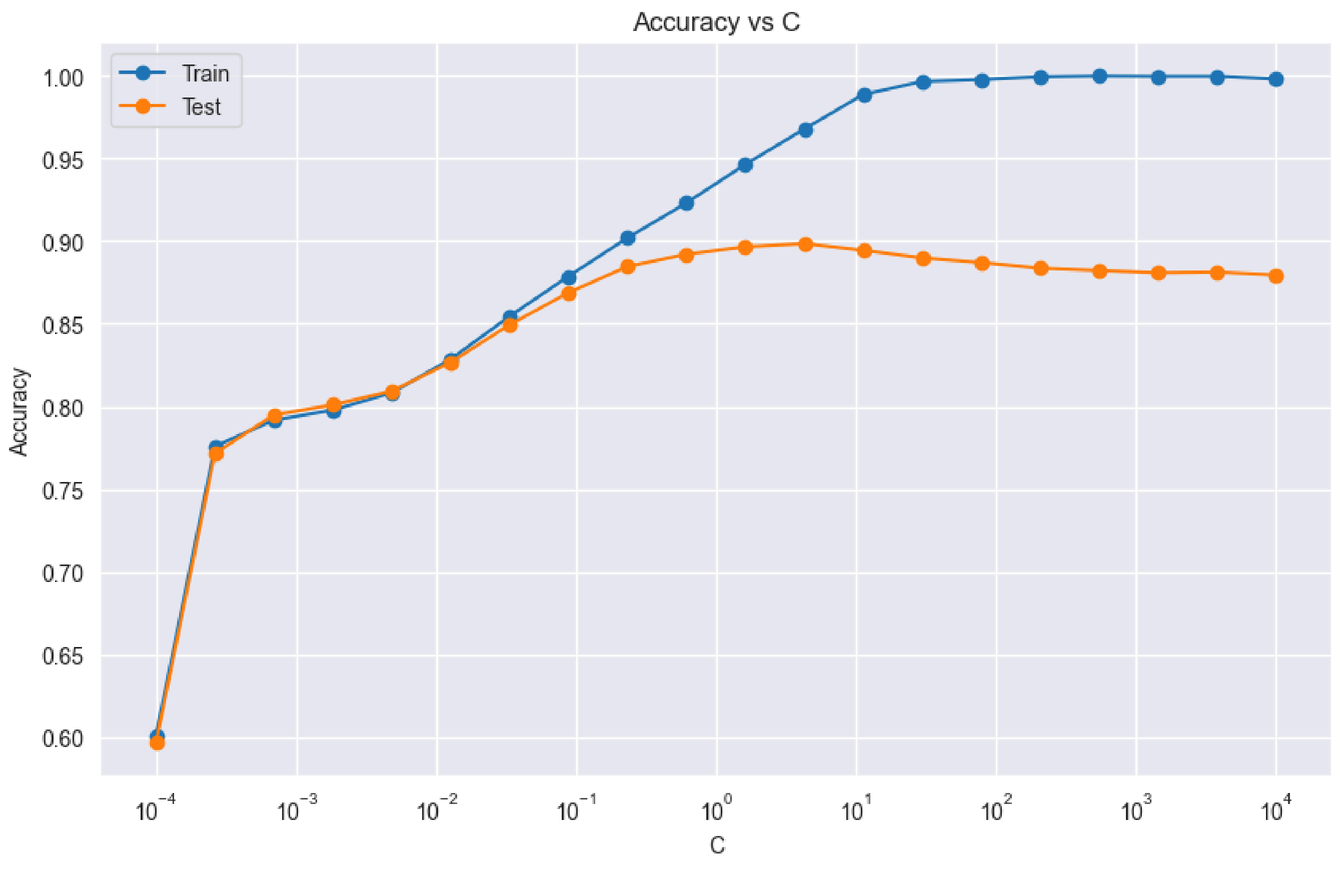
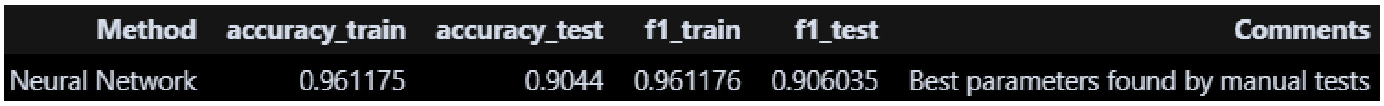


***Figure 4*: Logistic Regression Accuracy vs C evolution**

Neural Networks:

Tuned model:



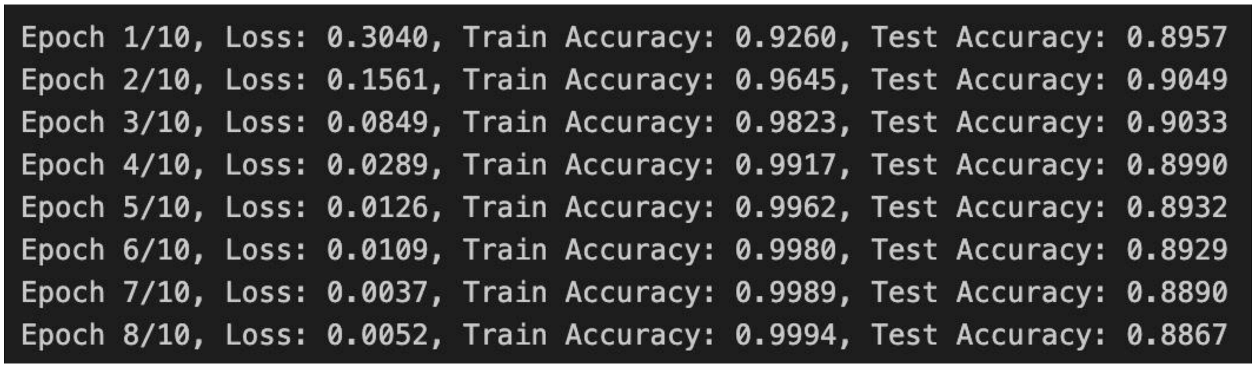
Method	accuracy_train	accuracy_test	f1_train	f1_test	Comments
Neural Network	0.961175	0.9044	0.961176	0.906035	Best parameters found by manual tests

The structure of the Neural Network is the following one:

```
        fc1 = nn.Linear(input_dim, 128)
        fc2 = nn.Linear(128, 64)
        fc3 = nn.Linear(64, 1)
        relu = nn.ReLU()
        sigmoid = nn.Sigmoid()
```

At first we expected better results, but after trying multiple different neural network structures, learning rates and number of epochs, this was the one that gave us the best results.
As was said in the presentation, maybe **adding layers that decreased more gradually the input size** (92692) would have **resulted in even better results**.

We tried just that, and after multiple attempts we could not get a model that did not end up overfitting. We tested different learning rates, batch sizes and even experimented with other methods to counter overfitting like **weight decay** but we were not able to get a model that did not heavily overfit.



```
Epoch 1/10, Loss: 0.3040, Train Accuracy: 0.9260, Test Accuracy: 0.8957
Epoch 2/10, Loss: 0.1561, Train Accuracy: 0.9645, Test Accuracy: 0.9049
Epoch 3/10, Loss: 0.0849, Train Accuracy: 0.9823, Test Accuracy: 0.9033
Epoch 4/10, Loss: 0.0289, Train Accuracy: 0.9917, Test Accuracy: 0.8990
Epoch 5/10, Loss: 0.0126, Train Accuracy: 0.9962, Test Accuracy: 0.8932
Epoch 6/10, Loss: 0.0109, Train Accuracy: 0.9980, Test Accuracy: 0.8929
Epoch 7/10, Loss: 0.0037, Train Accuracy: 0.9989, Test Accuracy: 0.8890
Epoch 8/10, Loss: 0.0052, Train Accuracy: 0.9994, Test Accuracy: 0.8867
```

The run times for these larger neural networks were way longer than those of our previous model (around 50 minutes for 8 epochs) and so we considered that we would just mention our attempt and conclude that the performance of our model could be improved by decreasing the input size more gradually but we did not have the resources or time to do so.

The performance of the submitted model might not be that much better than that of our logistic regression model because of the simplicity of our task, which just consists in dividing reviews in 'positive' or 'negative'. More insights can be found in the notebook and also in the conclusion of this report.

With that in mind, we still were able to find a configuration that outperformed the rest of our models.

The *figure 5* represents the evolution of the model.

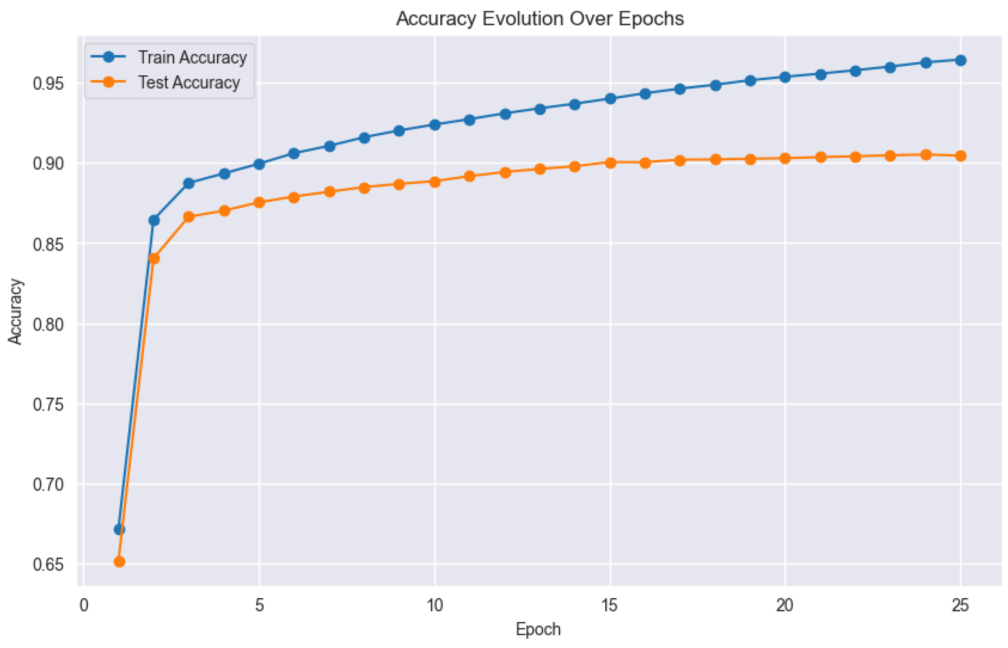


***Figure 5*: Neural Network Accuracy Over Epochs evolution**

Recurrent Neural Networks:

Tuned model:

Epoch [6/20] -
Train Loss: 0.3014, Val Loss: 0.3119, Train Acc: 0.8878, Val Acc: 0.8656, Test Acc: 0.8630

Epoch [7/20] -
Train Loss: 0.2875, Val Loss: 0.3026, Train Acc: 0.8962, Val Acc: 0.8744, Test Acc: 0.8693

Even though early stopping was used, we manually stopped the training process because it took more than an hour to reach epoch 7 and it only improved by 0.006 in the Test Accuracy.

We would have loved to have more time, and most importantly, computational resources to try different configurations as we could only try a couple structures and we couldn't even finish executing them as they took a huge amount of time to do so. We didn't ask for computational resources as this was a last time addition that we did but we'll try to improve it by ourselves just for curiosity.

Even though it's explained in more detail in the Notebook, to sum up, we have used the **LSTM approach** to solve problems that happen to raw RNNs such as **gradient vanishing and exploding**, we have used a **pre-trained GloVe embeddings** (downloaded from the Stanford Natural Language Processing group's official website) that gave the model a head start in the first epochs, and after some tests, we added the Bidirectional feature for the RNN with attention. Again, all of this is **widely explained in the dedicated part of the Notebook** to RNNs.

Model Testing

We considered that our model classifying reviews into either positive or negative was a bit too simple and we decided to go a step further and allow for our models to make movie recommendations from the reviews of that movie.

For this we implemented a **Gradio interface** which takes 10 text reviews as input (10 is just an arbitrary number) as well as a percentage that represents the minimum percentage of positive reviews the user expects among the ones introduced as input.

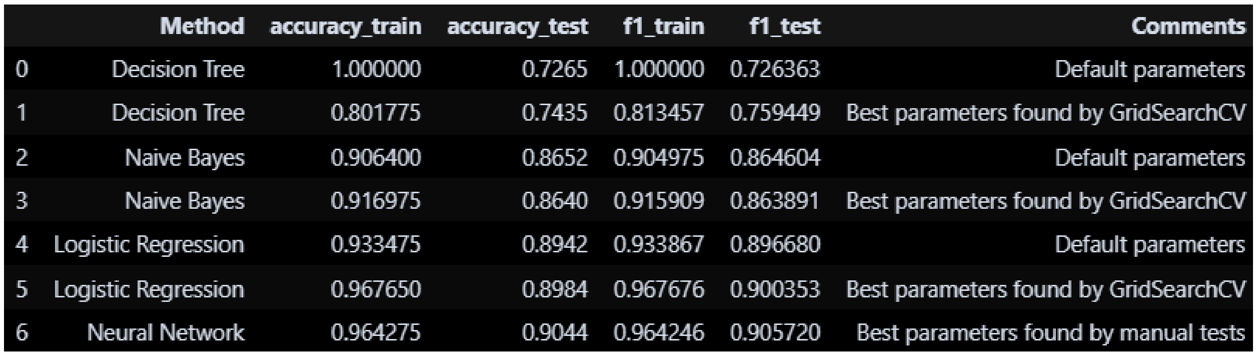
We also allowed to choose either Logistic Regression or Neural Networks to be used for this purpose as they were our two best performing models.

Screenshots for this interface were already shown during our presentation and can be found in our slides.

Note: if you need to try the live version of the Gradio website, feel free to contact us around 30 minutes before and we'll run everything that's necessary to generate a new fresh 4h valid link to the website that we'll email you back.

Conclusion

Here's a the data frame that sums up all the models performances throughout the notebook:



	Method	accuracy_train	accuracy_test	f1_train	f1_test	Comments
0	Decision Tree	1.000000	0.7265	1.000000	0.726363	Default parameters
1	Decision Tree	0.801775	0.7435	0.813457	0.759449	Best parameters found by GridSearchCV
2	Naive Bayes	0.906400	0.8652	0.904975	0.864604	Default parameters
3	Naive Bayes	0.916975	0.8640	0.915909	0.863891	Best parameters found by GridSearchCV
4	Logistic Regression	0.933475	0.8942	0.933867	0.896680	Default parameters
5	Logistic Regression	0.967650	0.8984	0.967676	0.900353	Best parameters found by GridSearchCV
6	Neural Network	0.964275	0.9044	0.964246	0.905720	Best parameters found by manual tests

The RNN result isn't shown as it was halted mid-training, but anyway it didn't achieve better results than the ones for the regular NN or the Logistic Regression model.
As we can see, both the tuned Logistic Regression model and the Neural Network achieve very similar results.

Logistic Regression performs really well in terms of text classification tasks, and even more when combined with strong feature representations like TF-IDF. It thrives in high dimensional, sparse feature spaces, and it's a simple linear model that can find a good linear decision boundary with proper regularization.
Also, as it uses a convex optimization it ensures to find a global optimum efficiently.

Neural Networks need more careful tuning and better feature representations. They are really good when there's nonlinear structures to exploit, or when there are richer representations such as embeddings (that capture semantic similarity between words). While using raw TF-IDF vectors, a Neural Network doesn't gain as much advantage from its representational capacity.

Decision Tree took a lot of time to train due to the high 'max_depth' values tested, and even with that we didn't get a decent result as a model. **Naive Bayes** took less time to train but its problem was that even with the tuning, it didn't improve a lot as the test accuracy plateaued in alpha = 0.1.

References

- Original Kaggle model
- Pandas Dataframe
- Sci-kit Learn Metrics
    - Accuracy
    - F1 score
- TF-IDF Vectorizer
- Decision Tree
- Naive Bayes
- Logistic Regression
- Neural Networks
- Recurrent Neural Networks
    - Bilateral Recurrent Neural Networks
    - GloVe embeddings