Machine Learning Project
Predicting shares of articles

Gauthier de Moffarts & Augustin d'Oultremont

1 Introduction

In this project, the goal was to predict the number of shares of an article on the web based on many components such as the day of the publication, the number of references to other articles or to itself or the rate of positive or negative words.

2 Preprocessing

Our code starts by reading the files and storing this data in pandas Data Frames. We then use a cyclical encoding (sin and cos of evenly distributed angles) rather than a one-hot encoding for the weekdays. This allows us to have 2 features instead of 7 and guides our models to understand that weekdays are cyclical. It also reduces the risk of facing the curse of dimensionality as it decreases the number of features.

The next step is to split the data between training and testing. This will allow us to give an accurate estimation of the score of our model. The (unknown) data of the testing set will only be used to assign a final score to our models. We did not use the testing set for our preprocessing as we have discovered that it is not recommended to fit a StandardScaler on the whole data instead of doing it only on the training data[1]. However, it is obvious that we applied all the transformations done on the training set to the testing set and to X2.

We used the train_test_split function. 20% of our data is kept as testing set and 80% as learning set. We decided to let it shuffle our sample before selection as we assume that the data have not been generated using a time-dependent process.[2]

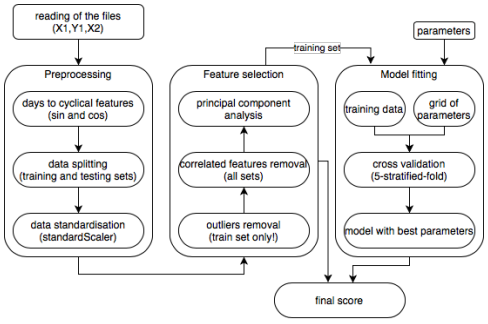

Figure 1: validation process

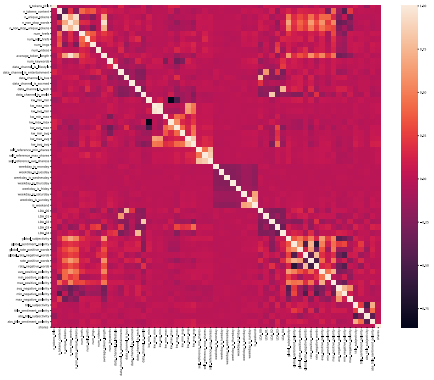

Figure 2: Correlation between features

[1] see https://scikit-learn.org/stable/modules/neural_networks_supervised.html#tips-on-practical-use
[2] see https://scikit-learn.org/stable/modules/cross_validation.html#a-note-on-shuffling

We then standardize our features by removing the mean and scaling to unit variance. It is done with the StandardScaler. We made this choice as we will use a PCA and other scaling sensitive models like the MLP.

We also followed the tip given in the project brief and removed the outliers using IsolationForest from sklearn. A usual run removes around 1000 outliers, representing about 5% of the dataset. This is certainly on the heavy side, but we the idea behind it is to help get a simpler dataset to work with, as scores are not very high and data is quite random. It is of course important to keep in mind that we work with a simplified version of the dataset and that the real world is much more difficult to predict, although our scores should be close to reality since we don't remove outliers from the testing set.

3 Feature Selection

We also detect the features that are too correlated with one another and for each pair of correlated features we remove the one that has the lowest mutual information with the target, thus limiting the redundancy. Doing so is beneficial for the KNN which would tend to attract the centroids more to redundant points of space if this precaution had not been taken. It also helps the PCA. We made a representation of the correlation matrix in figure 2 to emphasize the correlation between some features. This figure also tells us that all the features are about evenly correlated with the target (the shares, as can be seen on the last column of the matrix).

To remove even more features, we added a method to run a Principal Component Analysis (PCA) on our data and only keep a certain number of features. This number can be changed at the beginning of the code (`n_features_pca_kpca`) or in the function call. Note that it has not been used everywhere as we realised that the effect was not significant.

We also added a Kernel PCA. Indeed, we discovered that the Kernel PCA gives slightly better results than those produced by a PCA (see figures 4 and 5). This is not surprising as the kernel PCA goes to higher dimensions to apply PCA and can therefore find non-linear relations in our feature whereas the PCA can only find linear relations among them. However due to the fact that this function is very slow, we did not use it much during testing and preferred the PCA.

4 Model Selection

The function GridSearchCV is really helpful to tune the hyper-parameters of an estimator. We set its parameters in order to use a stratified 5-fold on our training data to avoid to overfit our data. We asked GridSearchCV to use the score_regression function given in the statement of this project to select our parameters as it is the function that we are asked to maximize.

In the remainder of this section, we will show how some parameters of the selected models influence the final score. As said in section 2, this final score is computed on the basis of the testing set which, it is important to point out, has been through all the modifications that X2 will undergo, but has not contributed to any information to obtain our models.

As you will see, every plot of performance is duplicated. We ran GridSearchCV twice for every test to diminish the possibilities of our conclusions to be drawn from random data.

4.1 K-Nearest Neighbors

When plotting the score against the number of neighbors (figures 3), we see that the 5-Fold test score stays around the same value, with only the training score decreasing fast. We seem to consistently get a score of 0.48 to 0.49, but no significant indication on the optimal number of neighbors.

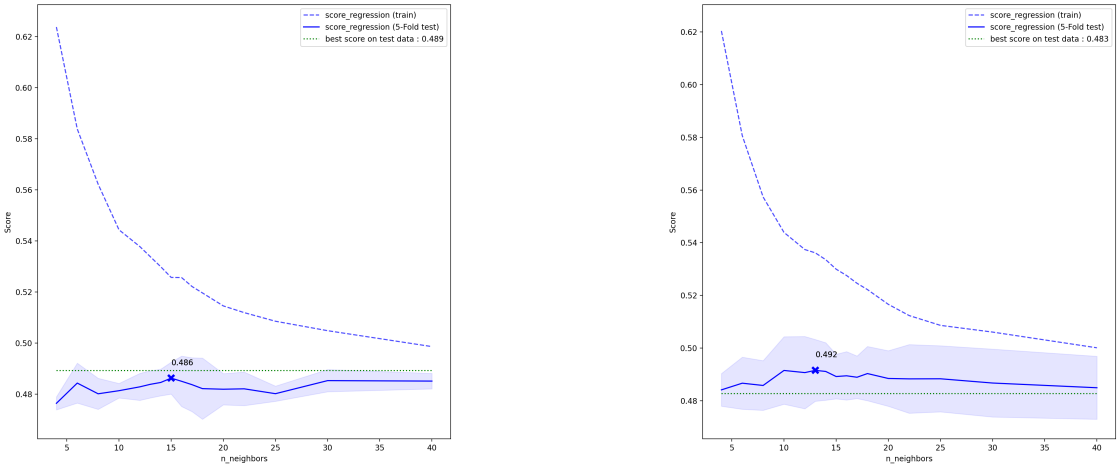

Figure 3: Score of a KNN with uniform weights

When using a PCA to reduce the number of features to 30, we did not get a significant difference in our results (see figures 4)and the number of neighbors does still have a minimal impact on the result.

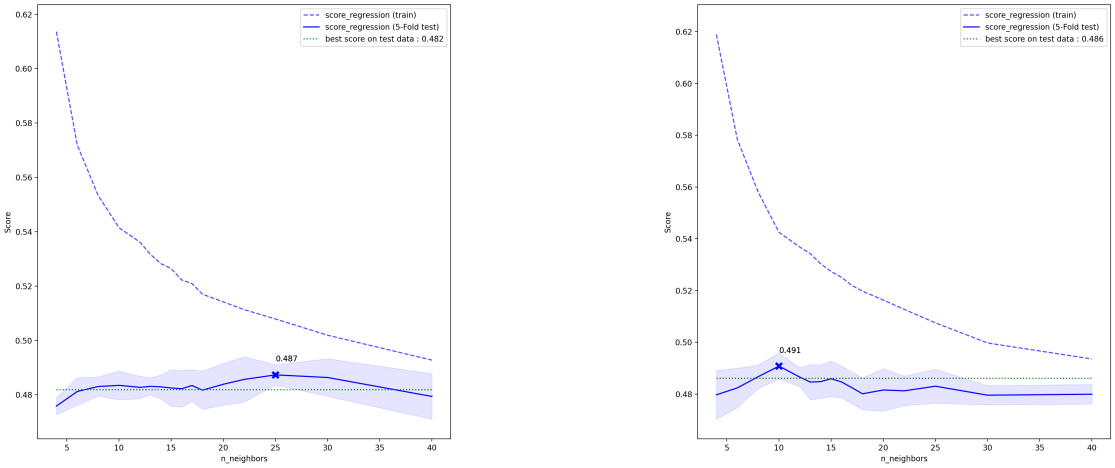

Figure 4: Score of a KNN (after PCA) with uniform weights

Using a Kernel PCA instead (see figures 5), we can see that the score on the 5-Fold test data gets a bit higher, but in the second case, the score on the actual test data is much lower. This is probably due to

chance since our dataset is very small.

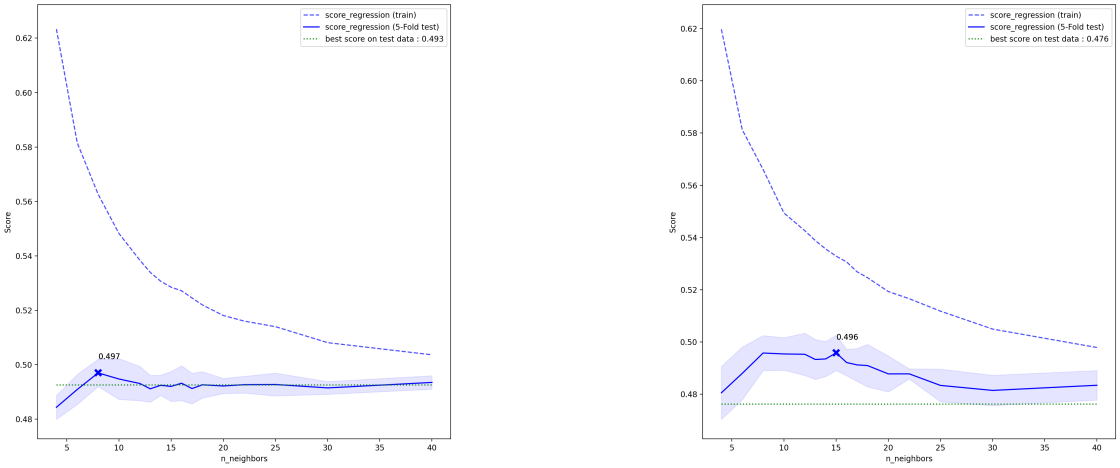

Figure 5: Score of a KNN (after Kernel PCA) with uniform weights

4.2 Multi-Layer Perceptrons

Using an MLP model gives better results, but only by a small margin (see figures 6). We can see that the model is better with around 3 layers of 25 to 50 perceptrons.

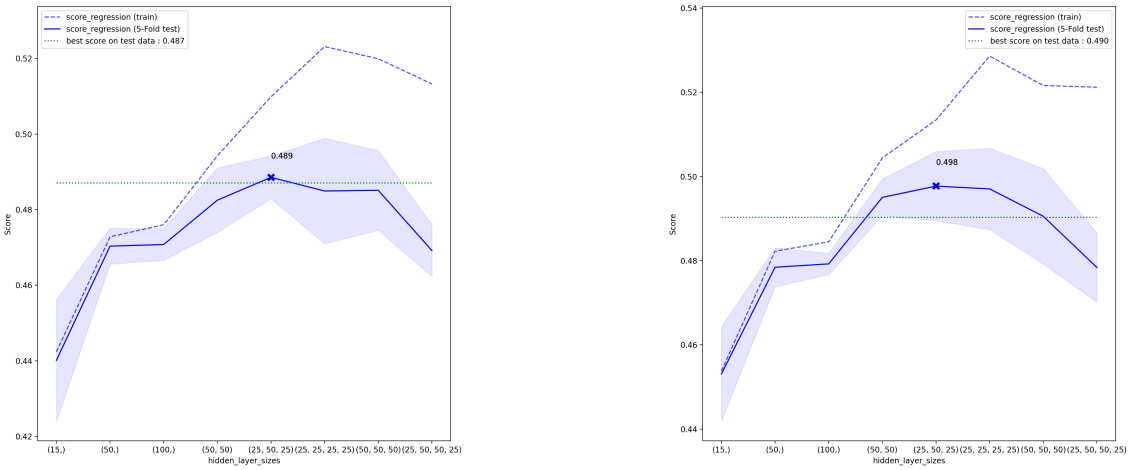

Figure 6: Score of a MLP with alpha set to 10^{-5} and an adaptive learning rate starting at 10^{-3}

Adding a PCA to restrict the number of features to 30 gives better results to high layer counts with lower layer sizes (see figures 7).

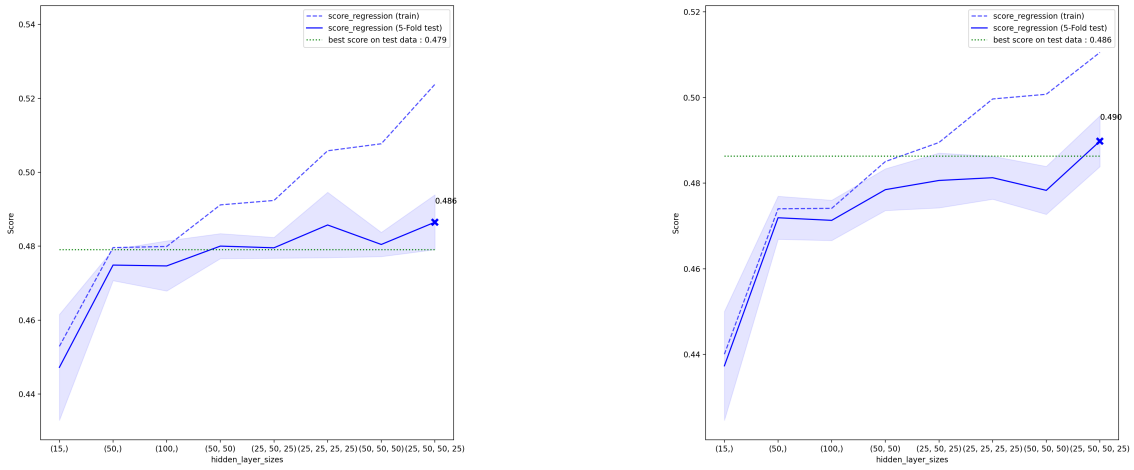

Figure 7: Score of a MLP after PCA with alpha set to 10^{-5} and an adaptive learning rate starting at 10^{-3}

4.3 Extra Trees Regressor

We can see that the ETR model is better than KNN and MLP (see 8). It achieves a score of around 0.50 to 0.51.

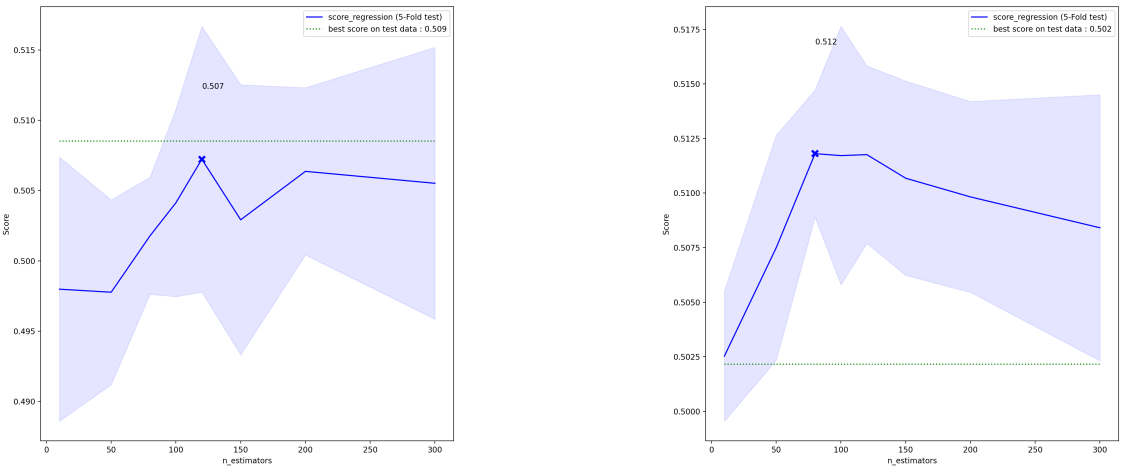

Figure 8: Score of the Extra Trees Regressor

To improve our score even more, we used the `sample_weight` argument when fitting the model. By setting this to the inverse of the number of datapoints in each category (flop, mild success, success, great succes and viral hit). This allows us to distribute the weight evenly between the categories. We can see that this has a small impact on the results.

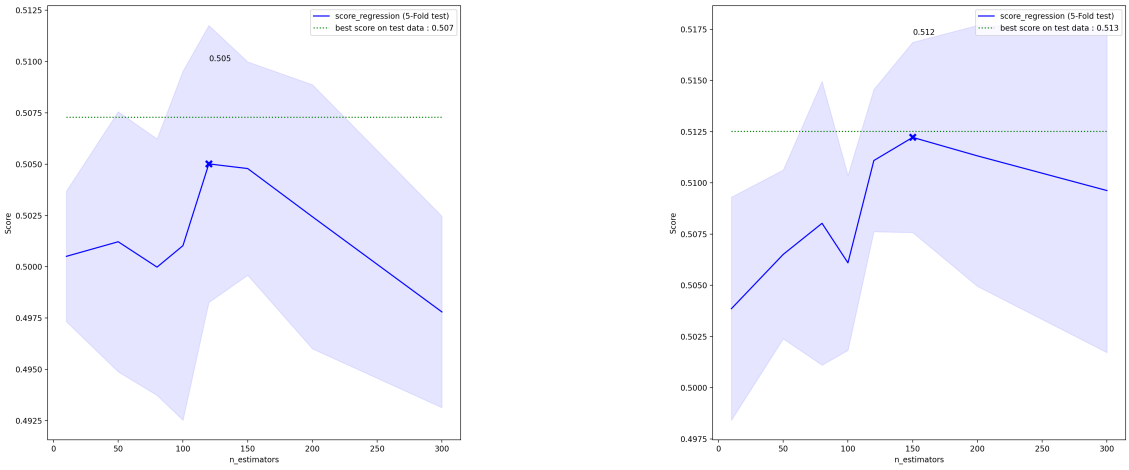

Figure 9: Score of the Extra Trees Regressor when setting the sample weights inversely proportional to the distribution

Considering that the Extra Trees Regressor model works well with high-dimensional datasets, we plotted the score obtained when running the model while keeping all the features and all the datapoints (keeping the outliers and not removing correlated features). The scores in this case are better (see figures 10).

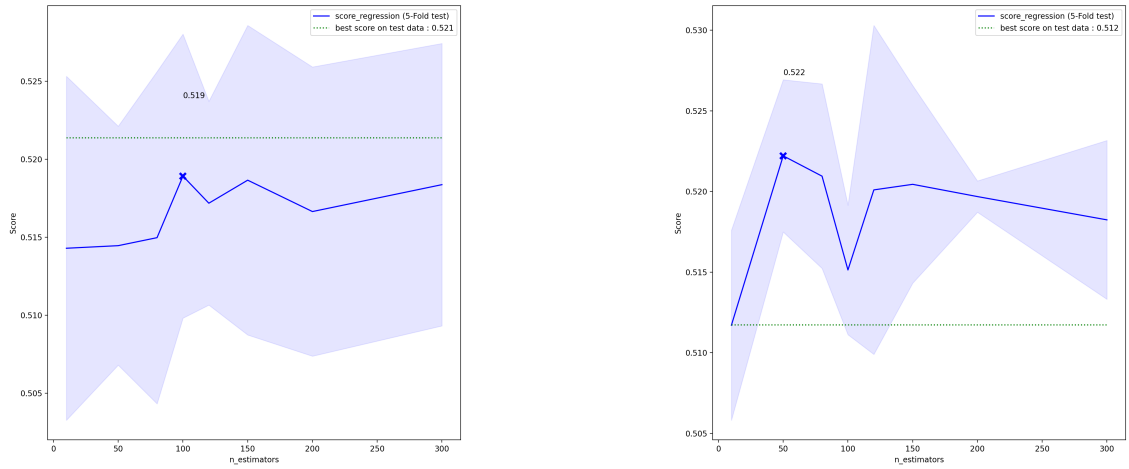

Figure 10: Score of the Extra Trees Regressor when applying only a small amount of preprocessing

5 Choice of the last model

As you saw in the previous section, we chose the implement an Extra Trees Regressor as our last model. We chose this model because we read that it is quite robust to noisy features and that it seems to perform quite well on our data. It is also usually faster than the random forests while keeping about the same

score.[3].

What we observed is that the model we chose does not need much preprocessing, and it works even better without. This is confirmed by this article. The same article mentions that decision trees overfit quite fast, but we use a K-fold which already reduces the chances of over-fitting, and the ExtraTreesRegressor fits a number of random decision trees averages them to improve accuracy and control over-fitting[4].

The decision trees also seem to have biases when the data is not distributed evenly, but the use of `sample_weight` allows us to counteract this flaw.

6 Conclusion

As is usual with machine learning projects, obtaining great results with the limited hardware and time we have at hand is quite difficult. The main purpose of this project was to learn about different machine learning models, and our web searches have lead us to learn much more than we could fit in this report.

There are of course still many paths to improving our model and there exist probably better models than the extra trees regressor. Setting aside the usual ways to improve our prediction such as having more data (which would require more processing power), we think that spending more time tweaking the different parameters and having a better idea of where the data comes from, how it has been handled and other information could help us get a better grasp on the dataset and the model.

From our point of view, tweaking all the parameters of the project to get to a maximum score is not the focus of this project, and it is much more interesting to have a general view of many models and have an idea of the general impact of the different parameters.

[3] see `https://www.thekerneltrip.com/statistics/random-forest-vs-extra-tree/`
[4] see `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesRegressor.html`