Supervised Learning - Practice Report

Aprendizaje Automático I

ELENA CAMPANERO BELDA
NATALIA KLINIK

1. Context.

The aim of this practice is to approach a supervised classification learning problem. We are given a dataset describing bank transactions, where the label column informs us whether the customer will be a debtor or not, judging by the set of bank transactions.

2. Data Preprocessing.

The first step was to perform data preprocessing, which from the beginning turned out to be problematic. The provided dataset is huge, which makes it impossible to preprocess it directly, as the "train_data.csv" weighs approximately 9GB and "test_data.csv" - 6 GB.

2.1. Split data into chunks.

We split the dataset into smaller chunks, reading 500,000 rows into each file. As the data is grouped by the client ID, we also made sure that the datasets contain all of the transactions assigned to the particular client. Thanks to that, we avoid having the same ID in separate files. We saved the new datasets into archives, to make future operations easier.

2.2. Checking rows and columns for NaN values.

Firstly, we check how many of the 189 columns have more than 50% of null values. We decided to delete them directly, as they would not give us valuable information. We also checked if there were completely empty rows, but we have not encountered any. After that, we created an MSNO Matrix, which was hard to read because of the number of columns, but gave us the idea of how the NaN values were behaving. Then we checked if there were any columns with 0 variance, as it would mean that they are filled with the same value in every row. No dataset would contain those.

2.3. Datatype analysis.

Secondly, we performed the datatypes analysis. The dataset contained variables of types "object", "float64" and "int64". We started with transforming the "ID" to a numeric variable, using the .to_numeric method, as it makes the comparison of IDs much easier than comparing strings. Then, we took the "Expenditure_AHF" column that contained the date-like data and transformed the date into a separate column for day, month, and year.

Later, we extracted other categorical columns and used LabelEncoder to transform them. Everything went well with the 'Infraction_YFSG'; however, the 'Infraction_DQLY' contained NaN values. As the percentage of those was around 4% in each data frame, we filled the NaNs with the most common value occurring in the data frame. Then, we performed the encoding. It turned out that 'Infraction_CLH', 'Base_67254', and 'Infraction_TEN' share the same labels. We created the frequency map and applied it to all three of the columns. Again, we encountered the NaN values. The percentage of those varied between 0.04% and 3.92% and, as it was very little, we filled the NaNs, this time using the mode.

2.4. Further NaN analysis.
We analyzed the rows and checked which of them contained more than 30% of NaNs. Turned out that it was less than 0.3%, so we could drop them. Filling them with other values would not make sense, as the rows would be completely artificial. Later, we could proceed to the remaining NaN columns. Those, where NaNs took more than 6% of the whole column, have been dropped, as filling it wouldn't make sense. For the remaining numerical columns, we took the following steps: if the standard deviation was lower than 1, we filled the NaNs with mean; if the standard deviation was higher than 1, we filled them with median; for the standard deviation equal to 0, we used mode. Finally, we checked if the data frames contained infinite values, but fortunately, they did not.

3. Feature Selection - first attempt.

When all the datasets were cleared, we merged the 'label' column with the rest of the data frames. Then, we performed feature analysis to select the features that describe the data in the best way. We used all three models presented during classes: filter, wrapper, and embedded.

3.1. Filter method.
The first method we used was the filter method. We used ANOVA to search for the 5 features that describe the dataset in the best way. We executed the code for each of the seven chunks and then took the interception of selected features. The code executed really fast, it lasted about 5-6s per dataset. The selected features were: 'Base_69608', 'Infraction_QJJF', 'Base_80863', 'Payment_6804', 'Base_85131'.

3.2. Wrapper method.
For the wrapper method, we started with RFE but as it is a recursive method it took forever to execute. We switched to RandomForest and obtained the desired result. The code was executing for about 2-3 minutes per data frame, which we accepted not to be too long. This method returned the same set of features for each of the seven datasets. The selected features were: 'Base_2810', 'Infraction_CZE', 'Base_85131', 'Infraction_QJJF', 'Payment_6804'.

3.3 Embedded method.
We started by using the Ridge Regression, which worked well, but took about 4-5 minutes per data frame to execute. The following features were selected: 'Payment_6804', 'Base_7744', 'Base_80863', 'Infraction_QJJF', 'Risk_9995'. Then, we tried the LASSO Regression and it gave us results in about 9 seconds per data frame. Here, we also achieved the same results for each dataset. The selected features were: 'Payment_6804', 'Base_80863', 'Infraction_QJJF', and 'Base_85131'. For the df_data_part_3 'Base_69608' was also selected.

4. Hiperparameters.

Then, we proceeded to hyperparameters tuning to optimize each model's performance metrics through grid search and cross-validation. As it was impossible to perform cross-validation with that many columns, we intended to reduce the dimensionality of our dataset as much as possible. We checked for columns with low variance and resulted in deleting the following: 'Base_5441', 'Expenditure_GCAO', 'Infraction_ZRH', 'Risk_8742', 'Risk_6197', 'Risk_3506', 'Infraction_PTY', 'Risk_9247', 'Risk_4160', 'Risk_5270', 'Infraction_KSBR', 'Base_23737', 'Base_7331', 'Risk_6178'.

As Grid Search is a recursive algorithm, using it did not make sense on such a big dataset. We tried performing SelectKBest and RandomForest, but those algorithms also did not result in being useful.
This code finished with a message: 'The optimal value of k is 8'. It is important to note that the program stops when the improvement is lower than 0.0000000001 points. Maybe it would be reasonable to state a less tight margin. When we had not specified the stopping point, the code was executing for too long and did not return anything.

Finally, we decided to use OPTUNA - an open-source software framework for automated hyperparameter optimization - to find the number of features we should use. We started with setting the feature limit between 1 and 30, and the code returned the maximum of those. We increased the limit to 60 and this time got k=53 as a result, meaning that it is the optimal number of parameters we should be using.

5. Testing different numbers of features.

Given that k=53 appeared to be a relatively high value, we considered creating another model with k=20 parameters. However, after a more thorough analysis, we concluded that the model's performance had already worsened with the use of 53 features and abandoned this approach.

On the other hand, we started to think if maybe using more than 53 features would make the performance better. We applied the same methods and tested the models that we were going to use in the practice while selecting k=100 features. The results were pretty similar to the previous one and, having in mind that increasing the number of features increases the time of the execution, we decided to continue with selecting 53 features.

6. Feature Selection - second attempt.

After choosing the number of features to select, we had our second attempt at feature selection. We applied the same models as before, this time selecting 53 best features for each dataset. Later, we extracted the intersection of all of the sets.

For the filter method, we decided not to use chi-squared, as it does not accept negative values, which occur in some of the columns. We have used ANOVA instead. Then, we used the Mutual Information Classifier and, as earlier, got the intersection of the 53 best features in all datasets.

For the wrapper methods, we used Random Forest and SelectFromModel RidgeCV. The results returned by each differed a little.
When it comes to the embedded methods, as we have tested earlier, the LASSO Regression lasts way less than the Ridge Regression, so we decided to only use the first one.

7. Results evaluation.

7.1. Motivation.

To evaluate our models, we defined several functions that analyze and visualize the results. We conducted the evaluation using precision, recall, and f1-score. As the aim of this practice was to analyze the bank data, we decided that the values that should be given particular attention are the false positives, meaning corrupted clients (label = 1) classified as not corrupted ones (label = 0).

We focused on checking the f1-score, remembering that we would also like to heighten the precision. A total number of values marked as positive 0 labels increases the percentage of true positives and, at the same time, decreases the percentage of false positives.

7.2. Profound analysis.

The general idea was to train various models for each feature selection method and each set of features selected. We used Decision Trees, Random Forest, Logistic Regression, SGDClassifier, KNN, AdaBoost, and Simple Neural Network to choose the best feature selection method.

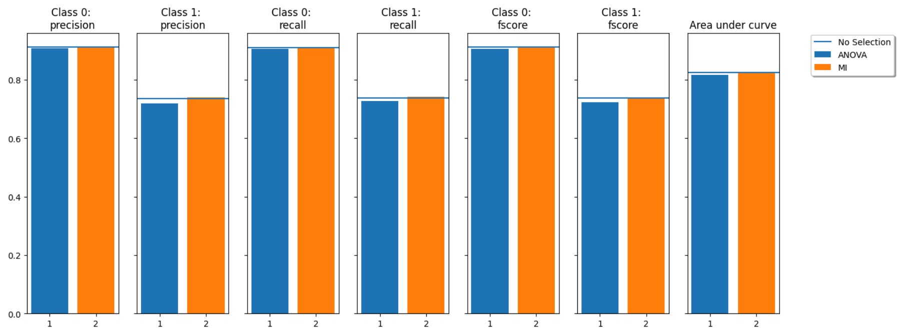
For the filter method, we tested ANOVA and Mutual Information Classifier; for the wrapper method RandomForest and SelectFromModel RidgeCV; and for the embedded method, we stayed with Lasso Regression.

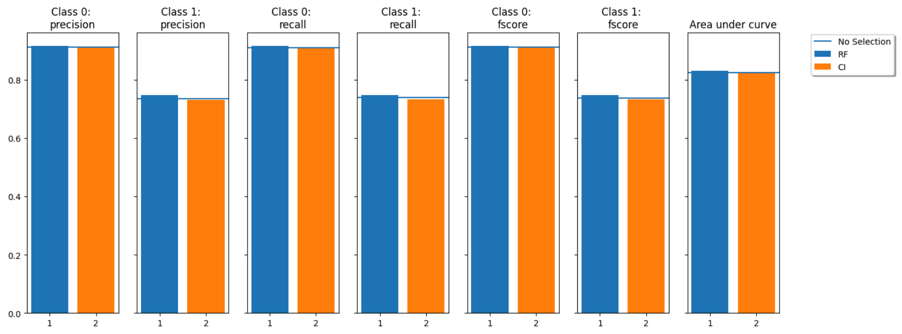
We used the plots to see how the values of precision, recall, and f1-score vary for each method. We used the scores achieved by a model with no features selected as a comparison. We separated data by classes, meaning label=0 and label=1, and, as scores for non-corrupted clients stayed stable, we only focused on analyzing the corrupted ones (label=1).

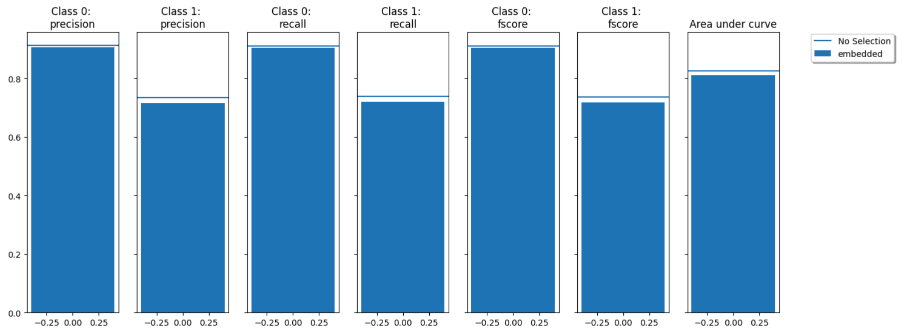
7.3. Evaluation of each model.

We plotted the results achieved by each model for all methods and checked how the f1-score changes.

7.3.1. Decision Trees.

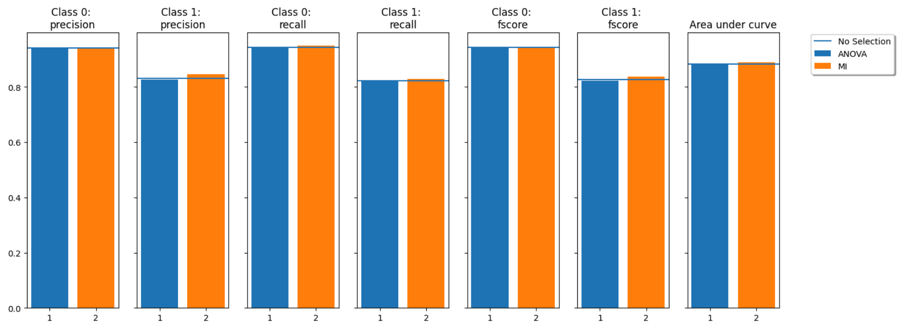


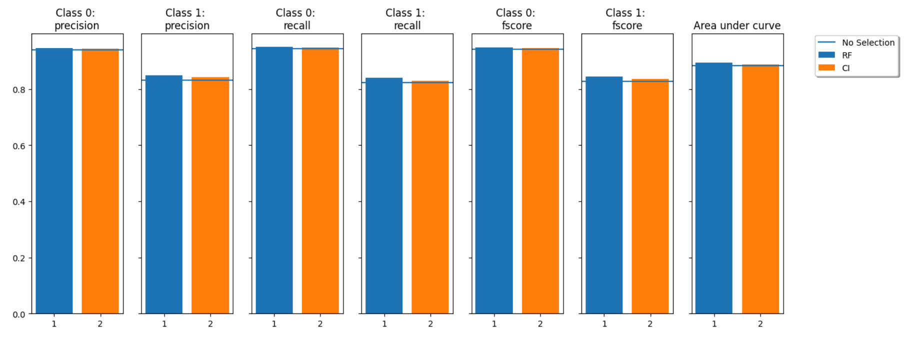


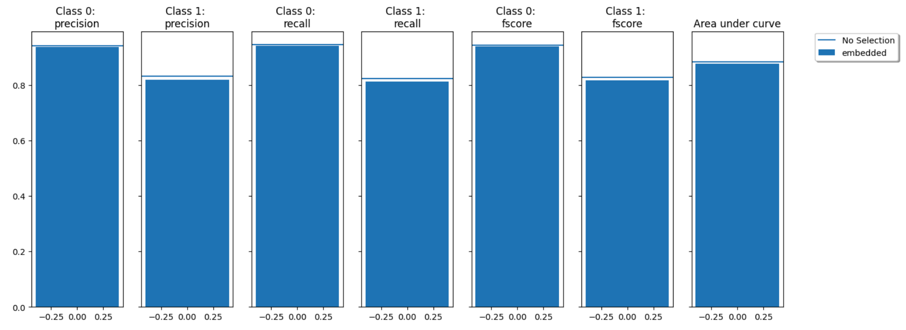


When it comes to the filter method, MI works better. With the wrapper, the results are very similar, but RaF is slightly better. The embedded method works the worst.

7.3.2. Random Forest.

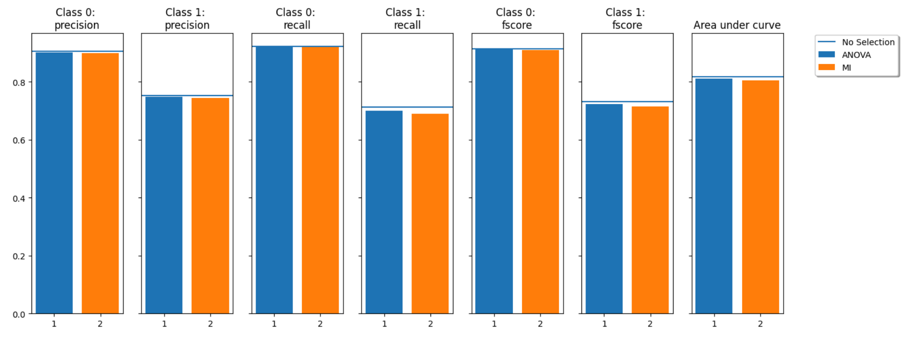


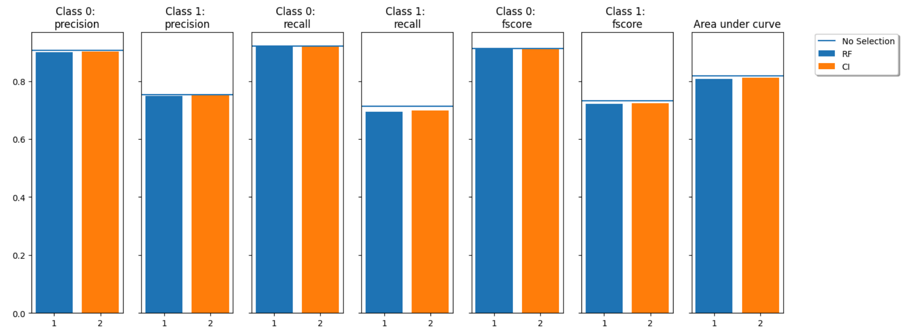


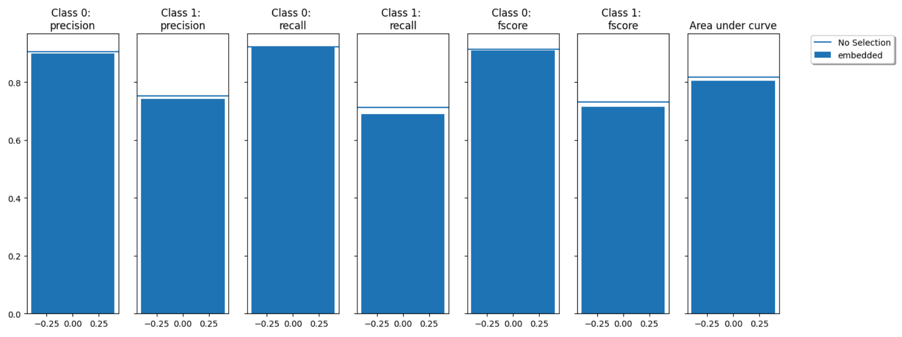


With the filter method, MI gives better results. RF is preferable for the wrapper method and the embedded method works well.

7.3.3. Logistic Regression.

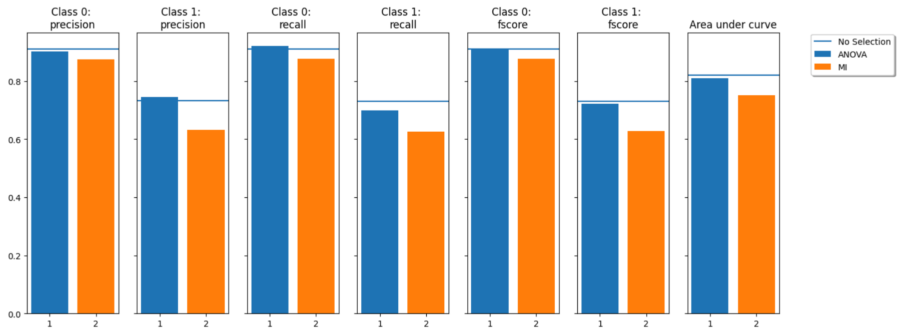


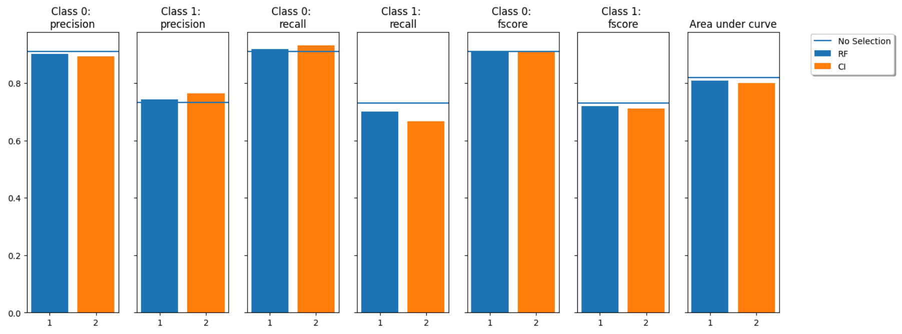


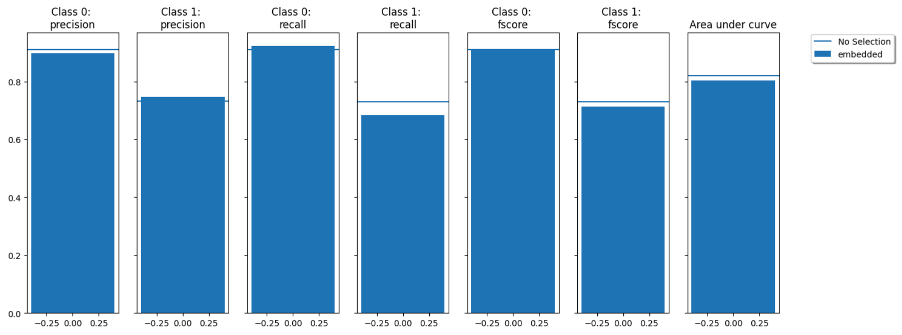


Filter methods give similar results in favor of ANOVA. For the wrapper method, CI works better. The embedded method works probably the worst of all.

7.3.4. SGD Classifier.

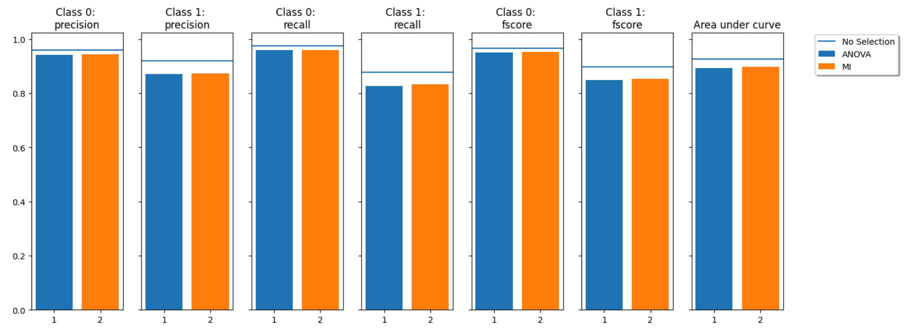


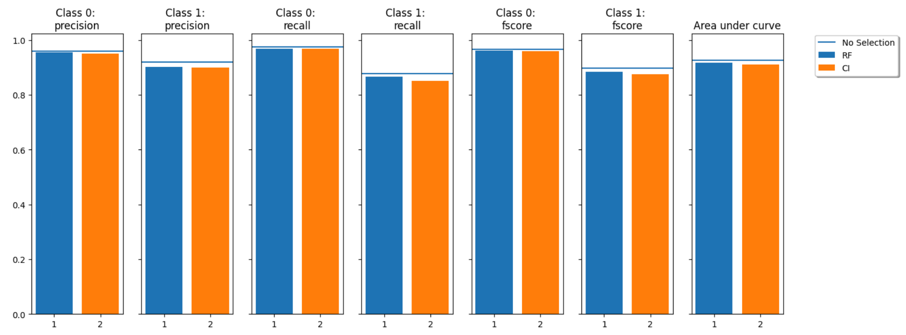


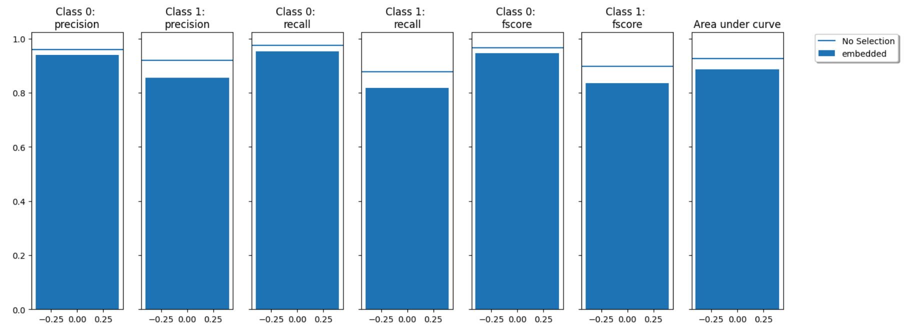


In this case, MI works really poorly. For the wrapper, RF works a little bit better than CI. The embedded method works well.

7.3.5. KNN.

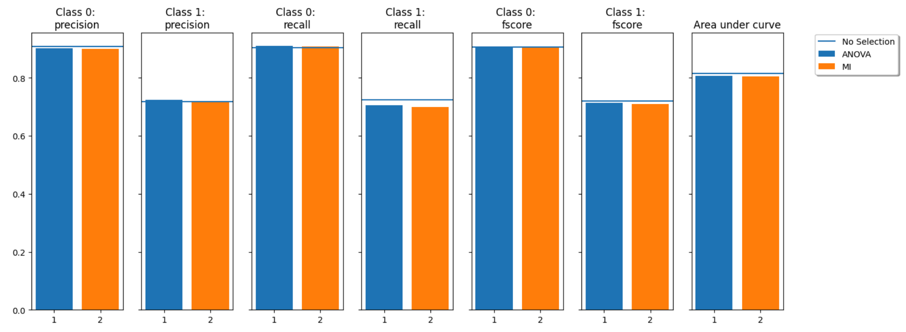


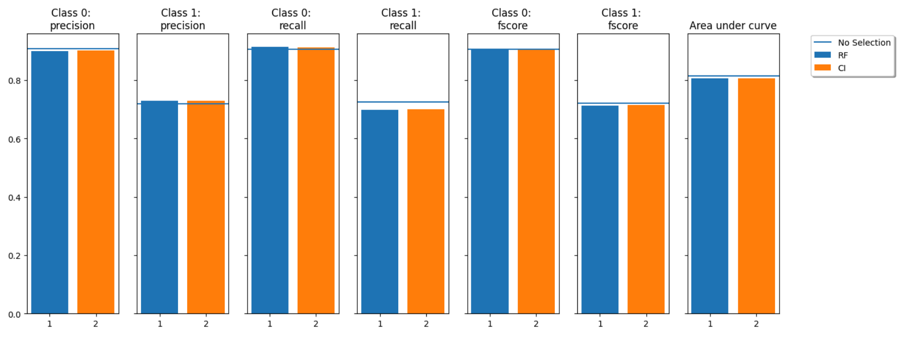


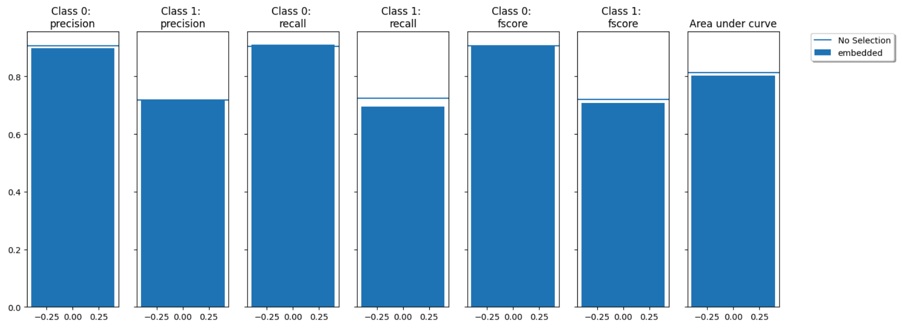


There is a visible drop with all of the methods, especially filter and embedded ones. For the wrapper method, RF works best.

7.3.6. AdaBoost.

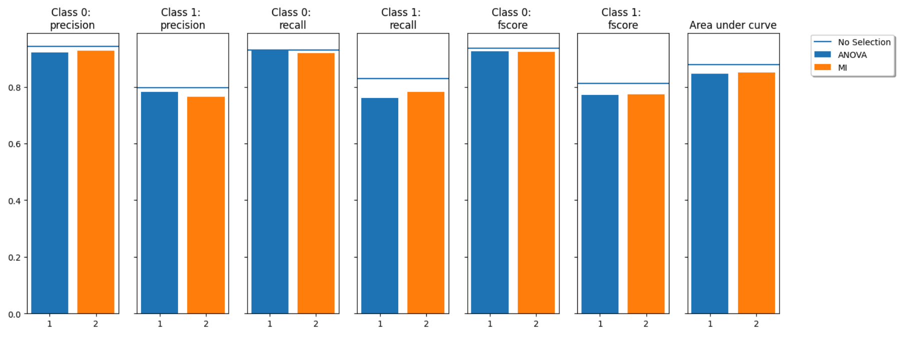


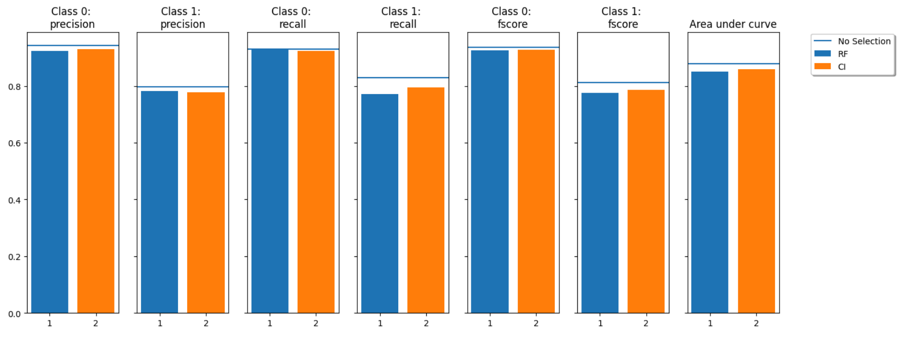


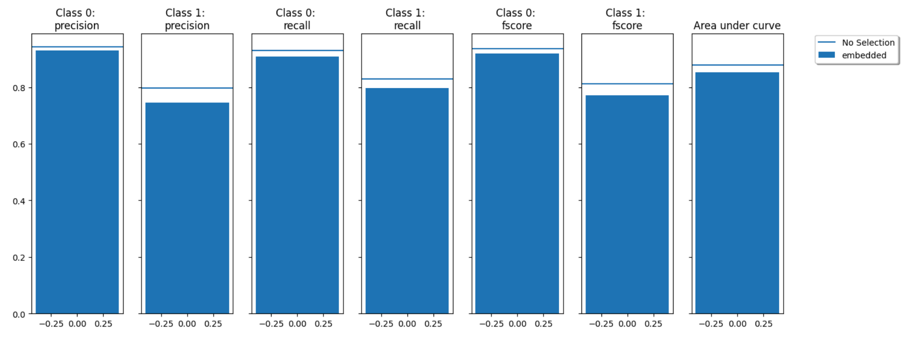


Here the results are pretty similar for all the methods. CI is probably the best one.

7.3.7. Simple Neural Network (fully connected).







There is a loss visible for all the methods. Again, CI gives the best results.

7.3.8. SVM Classifier.
When it comes to SVM, we were not able to achieve any results, as the code did not execute.

7.3. Conclusion.
We observed the results first while using 53 features and later with using 100 of them. As we decided before, we will stick with the lower number, as the results do not change much.

No matter what feature selection we are using, it worsens both the time of the execution and the general results. Each model works best with a different feature selection method and it probably would be logical to use a different method for each of the models. However, for the sake of this practice, we decided to choose one that we will work well with all of them.

We decided on the Importance Coefficient (CI), using SelectFromModel RidgeCV.

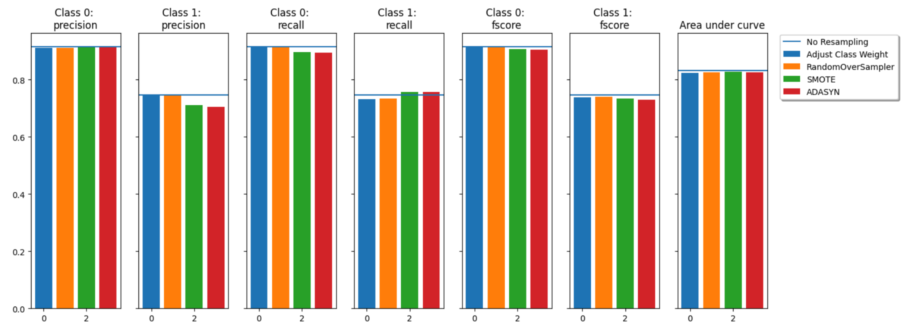
8. Imbalanced data.

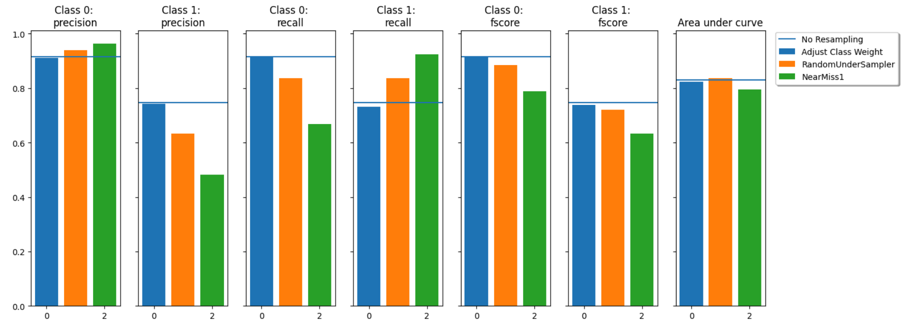
The data we were left with was imbalanced ('label' equal to 0 dominating over 'label' equal to 1), so we decided to apply various techniques to deal with that problem. We defined functions that measure and visualize the results. Neither undersampling nor oversampling appeared to be a good option. The first one would cut too much data, whereas the second one would make our dataset gigantic. We tried to perform our own method that would perform both algorithms, one after another, in order to achieve "in-between" results.

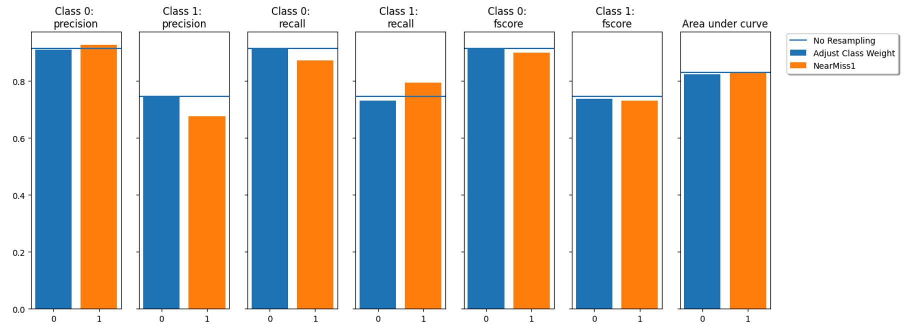
8.1. Analyzing the results.
When it comes to oversampling, we used RandomOverSampler, SMOTE, and ADASYN. For undersampling, we used RandomUnderSampler and NearMiss. We also tested the mixed method. All results have been compared with original data (before resampling) and Adjust Class Weight.

Again, we tested all the possible configurations with the same models as listed earlier.

8.1.1. Decision Trees.

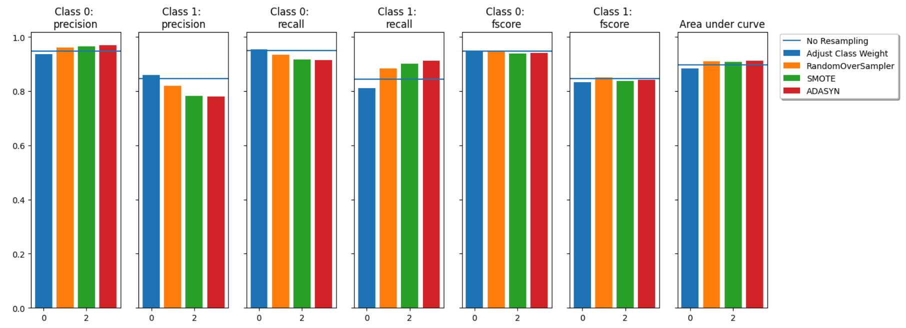


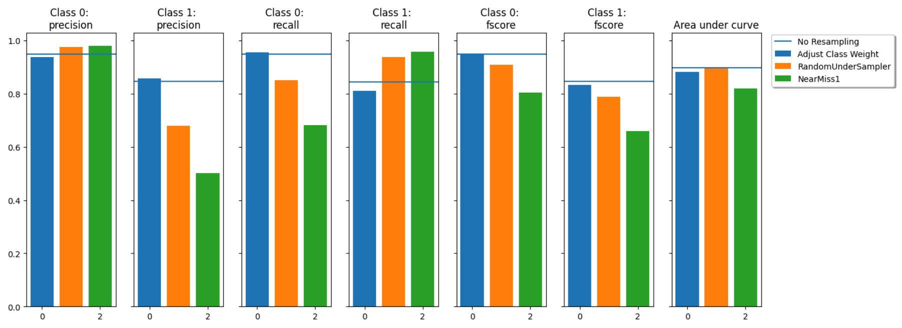


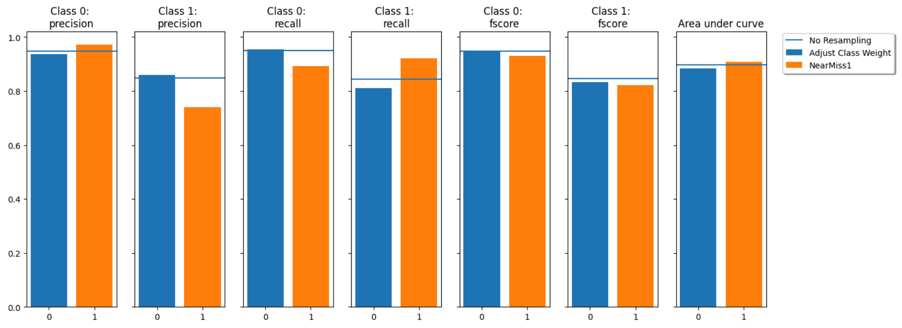


In this case, the best results were achieved by RandomOverSampler, AdjustClassWeight and our own "mixed" method.

8.1.2. Random Forest.

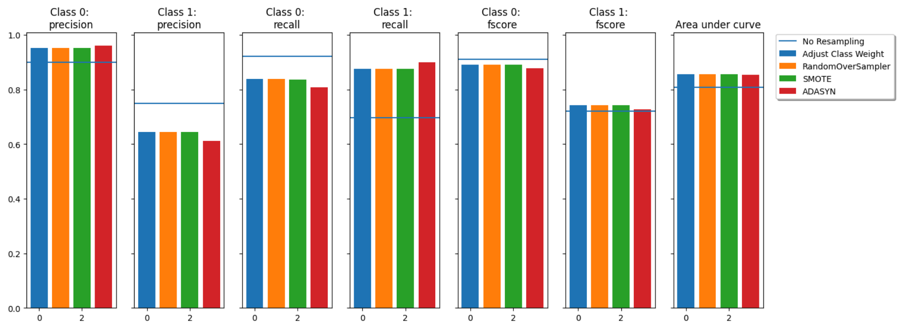


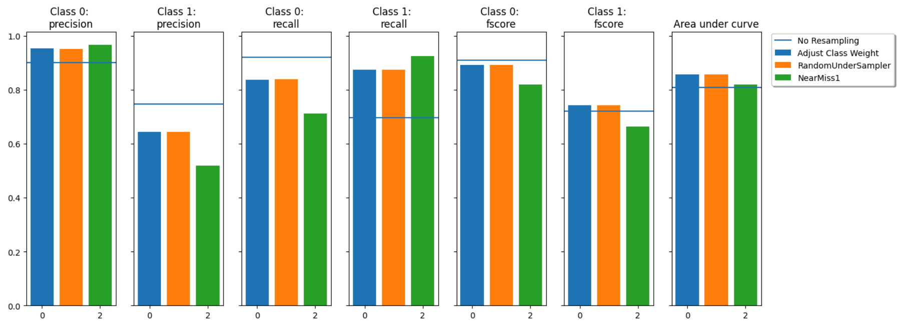


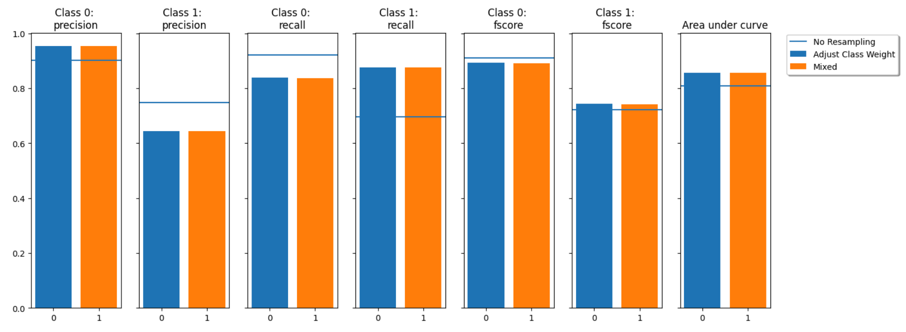


The best method was probably the RandomOverSample. AdjustClassWeight gives worse results, as well as the undersampling methods. The loss of our method is visible, yet not that bad.

8.1.3. Logistic Regression.

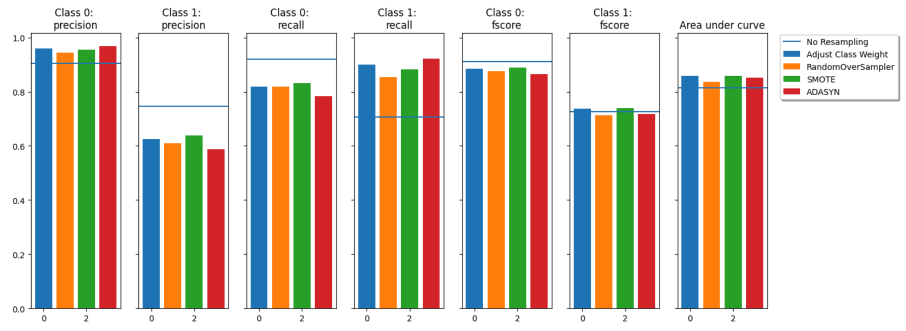


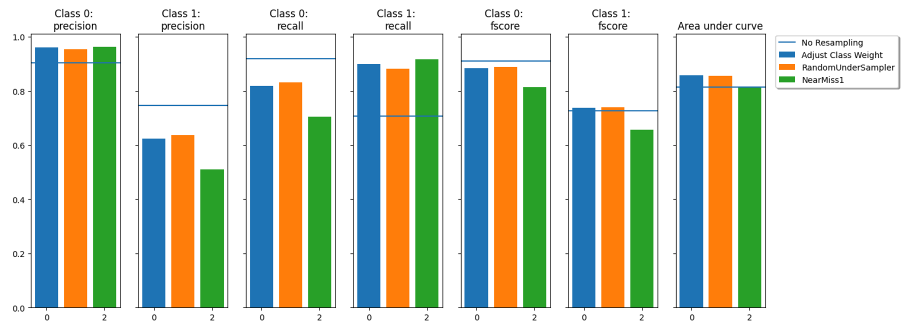


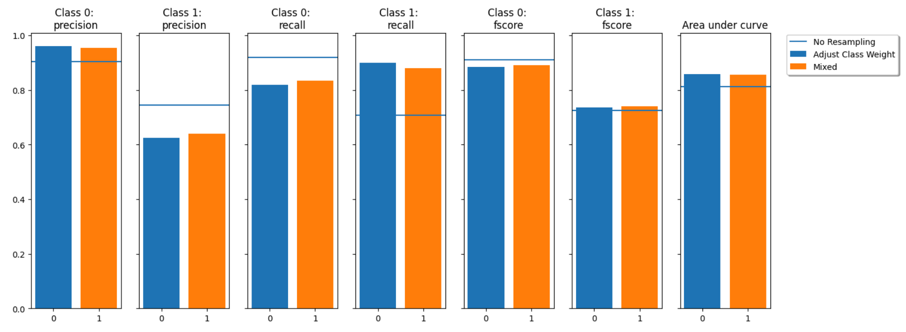


We could use any of the methods. NearMiss1 and ADASYN gave the worst results.

8.1.4. SGD Classifier.

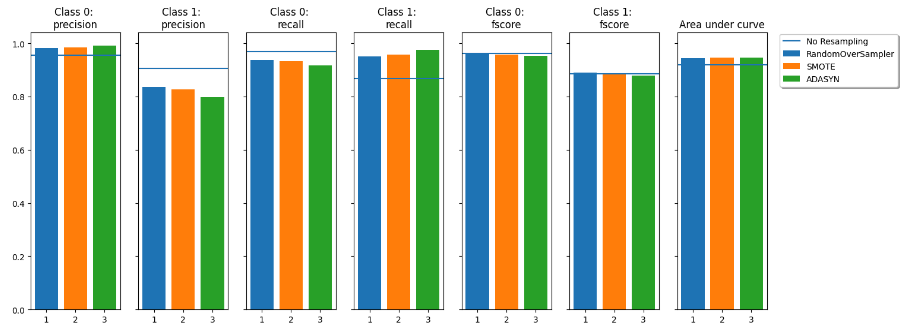


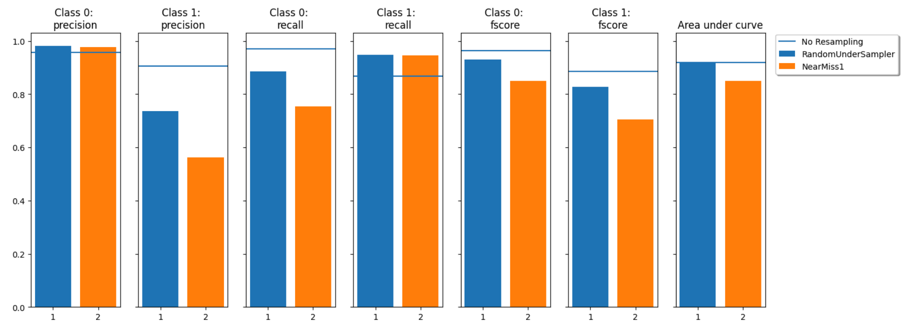


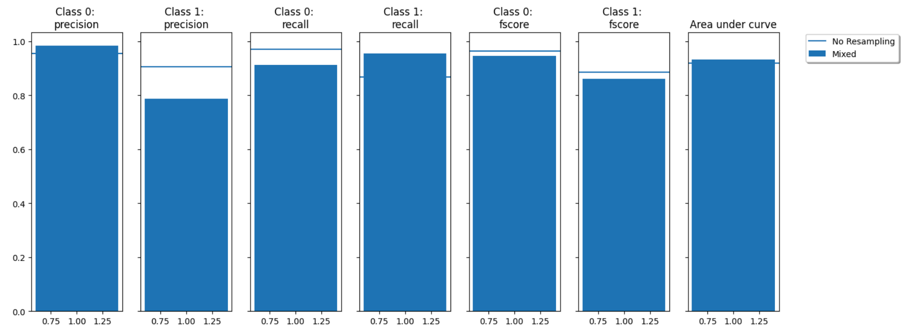


The results were similar for all of the methods, however the NearMiss seemed to be the worst.

8.1.5. KNN.

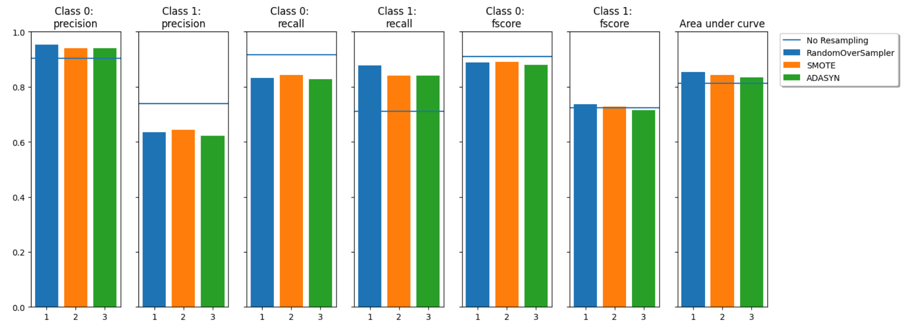


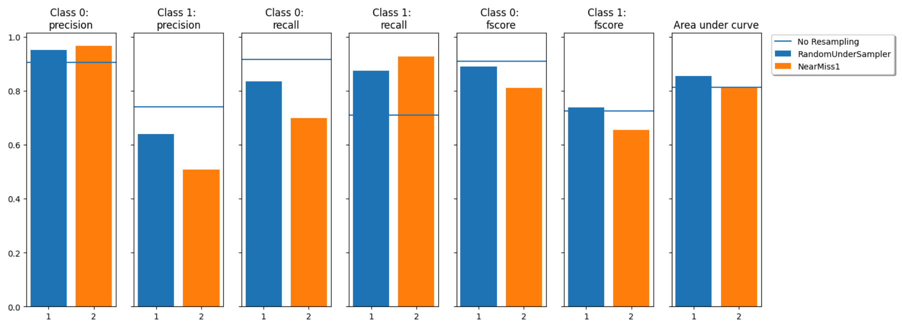


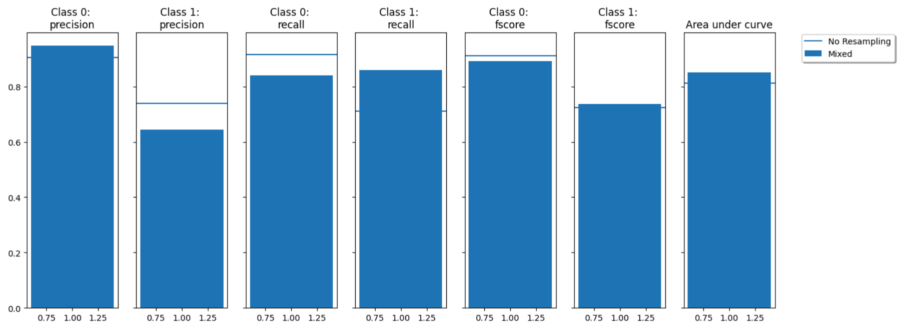


In this case, oversampling worked well. We would not recommend using the undersampling, the mixed method is better in this case.

8.1.6. AdaBoost.

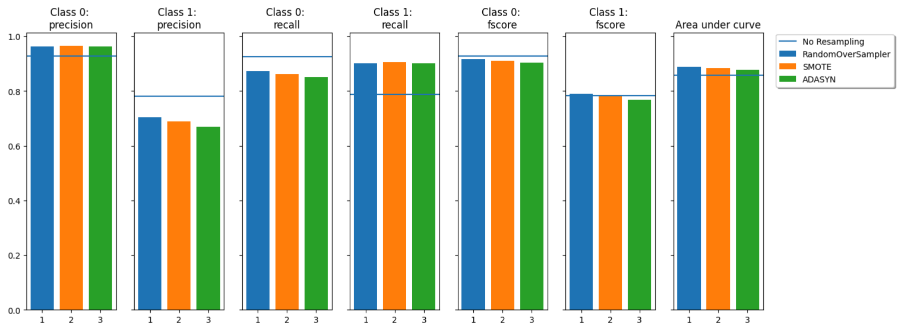


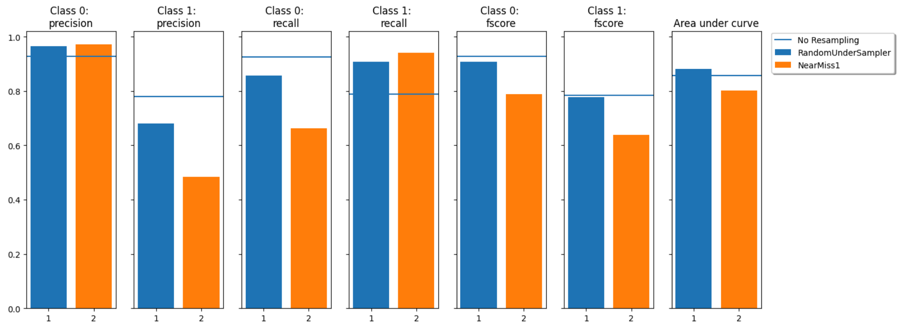


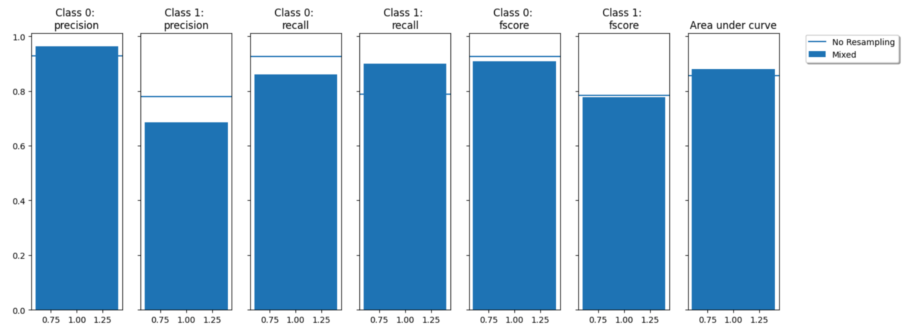


Undersampling did not really work in this case. It would be best to oversample the data or use the "in-between" method.

8.1.7. Neural Networks.







Here, it would probably be best to use RandomOverSampler or SMOTE.

8.1.8. SVC Classifier.

We intended to use the SVC Classifier as well, however the code took too much to execute. We believe that the other methods already provide us with all the necessary information.

8.2. Conclusion.

We should note that the improvement seen while using the oversampling methods can sometimes be false, as this method might easily lead to overfitting.

After the analysis, it seems that using overfitting would be the best option, however. We understand that it will obviously increase the size of the dataset, adding approximately 630.000 extra rows. If training the algorithms poses significant issues due to the size, we will divide the dataset one more time to extract the additional data frame (df_part_8). Thanks to that, the processing of each part will be reduced.

Another option would be to use undersampling. However, as we have observed, this method has worsened the results in all of the cases. If the size reduction is necessary, it will probably be the best option to use the mixed method.

9. Final Dataset.

9.1. Feature selection and balancing the dataset.

After profound analysis, we decided to use the Importance Coefficient for feature selection, as it gave us quite optimal results for all of the models. When it comes to data balancing, we decided on oversampling the dataset, using RandomOverSampler.

9.2 Preprocessing of the test data set.

When the train dataset was preprocessed, cleared and balanced, we proceeded to the test dataset. We followed similar steps: checked NaN values, performed data type analysis and dropped all the columns that we dropped with the train dataset. After that, we were left with two cleared dataset, with the same set of features, ready for the training.

9.3. Partition into test and train.

It is important to note that the oversampling should be performed after dividing the dataset into train and test subsets. In this way, we will avoid training artificial data.

While splitting the data into train and test datasets, we kept in mind to stratify the results. It means that we made sure that there is no thorn between transactions of the same client. This means that all of the transactions of one client belong either to a training or testing dataset. Thanks to that partitioning, there will be no problems with training the data.

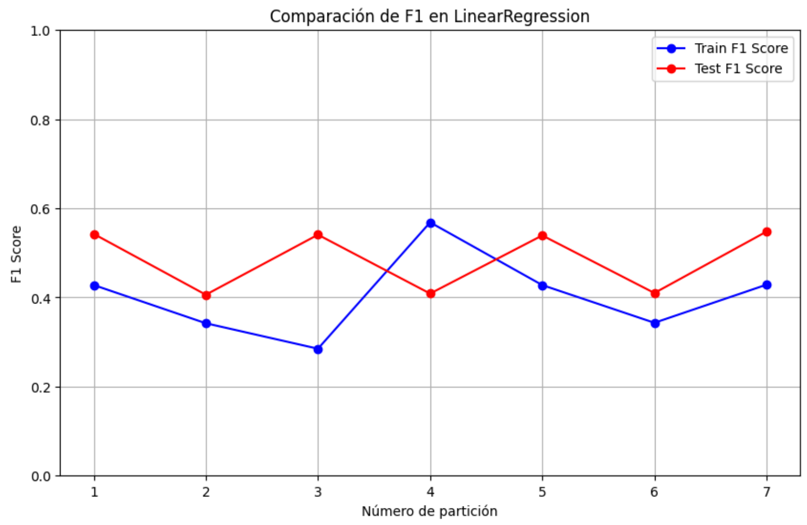
We decided on an 80% – 20% partition. It was performed on each of 7 initial data frames, resulting in 14 CSV files – 7 training and 7 testing ones.

10. Searching for the best model.

When our data was prepared and properly saved, we proceeded to supervised learning. As some of the models required normalizing the data, so we have done. We also defined several functions to help us evaluate the models. First, we created them and saved them in the .joblib format, as it facilitated later executions.

10.1. Linear Regression.

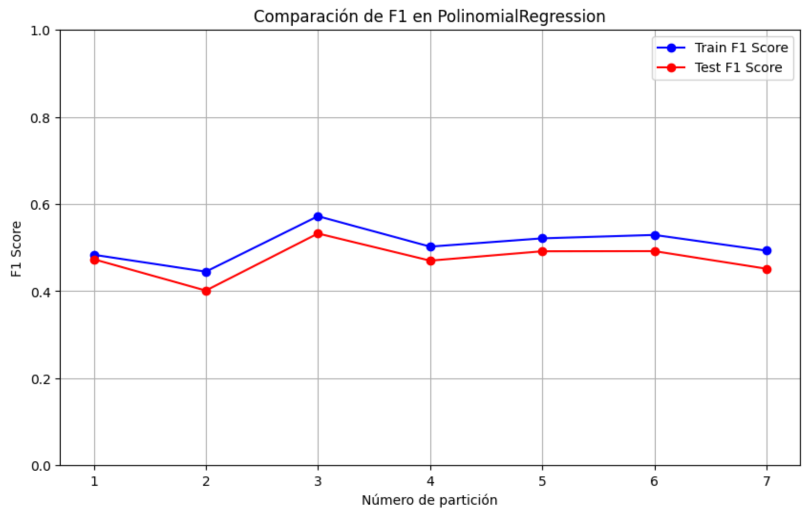
We have 7 training and 7 testing datasets and because of that, we have to use the partial_fit. As LinealRegression does not support that, we decided to use SGDRegressor with a 'loss' parameter set to mean squared error. This way, we achieve the same, rather poor, results, minimizing the squared error.



10.2. Polynomial Regression.

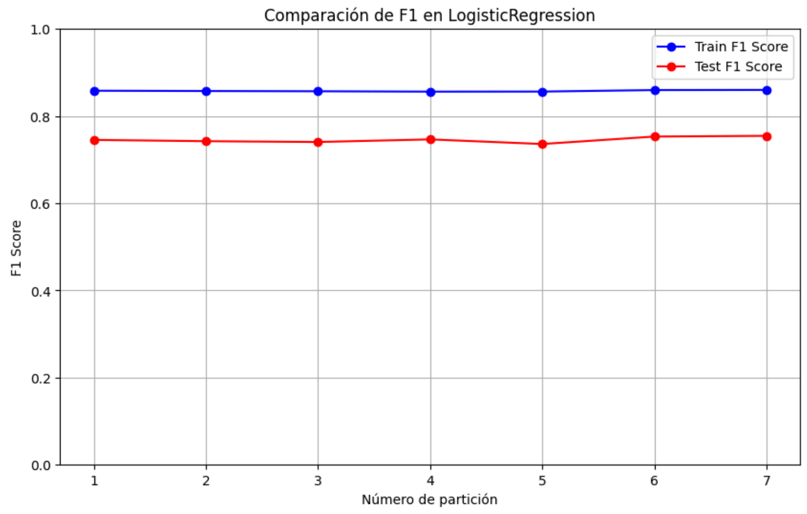
The problem with this model was that even while setting the degree to 2, the generated matrix was too big to be processed by SVC. We decided to perform the Principal Component Analysis first, to see if we could minimize the size of our model. As a result, we learned that the optimal number of components is 33, and that is the number we applied. We used PolynomialFeatures and SGDRegressor.

Again, the results were not great:



10.3. Logistic Regression.

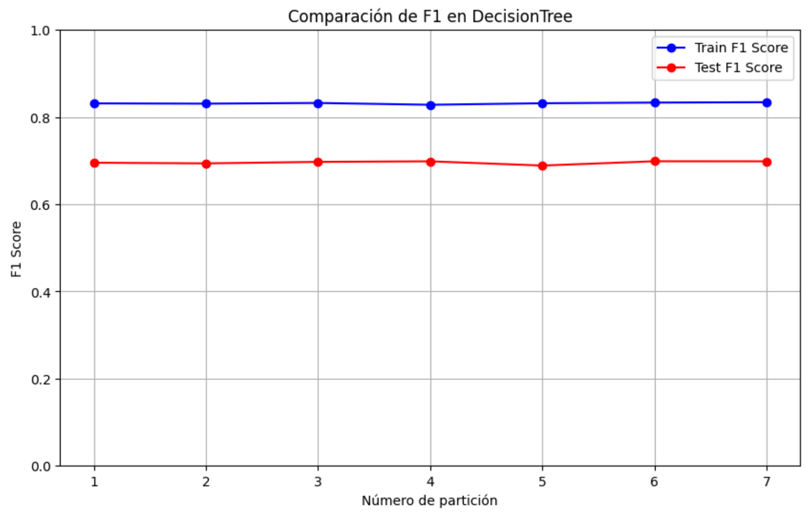
In this case, there was no need to additionally process the data. We used SGDClassifier with a 'loss' set to 'log_loss'. In comparison to the previous ones, the results were better:



10.4. Decision Trees.

Decision Trees do not support the partial_fit, so we decided to generate a separate tree for each partition and then apply an ensemble. We used min_samples_leaf=1 because it is usually the best choice for classification with few classes.

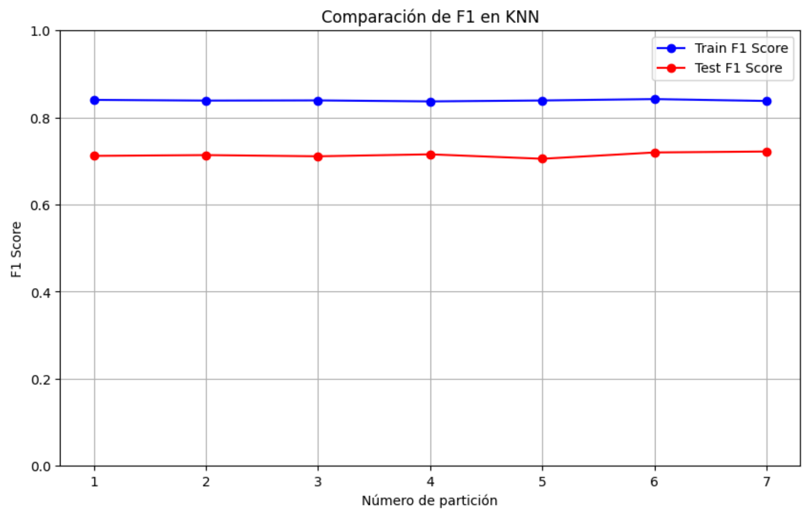
Later we used the VotingClassifier, which does a 'majority vote' of the trees separately to get the final prediction. Setting a 'soft' parameter indicates that it uses the average of probabilities instead of the majority of votes. The results were pretty similar to the Logistic Regression:



10.5. KNN.

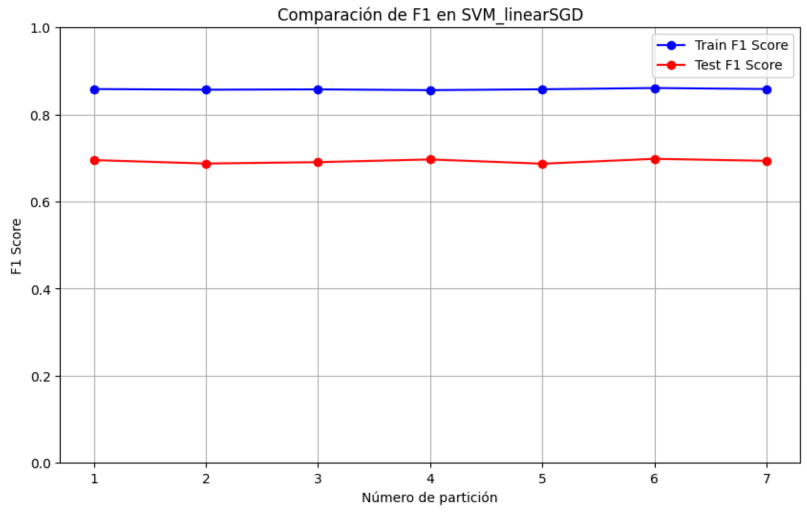
We have tried to use the 'annoy', which, according to the documentation, helps to approximate KNN with a memory-efficient approach. However, even after using it, the kernel died trying to train the model.

As no other solution worked, we decided to take a small representative sample and train the model with it. It gave the following results:



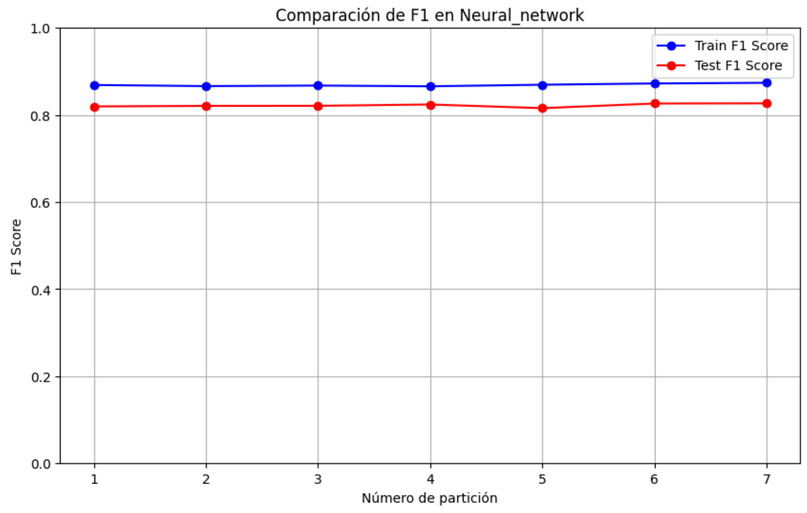
10.6. SVM Classifier.

Since SVC does not support the partial_fit, we decided to try using SGDClassifier with a loss='hinge' to simulate a linear SVM. We have also tried to do a PCA before running SVC but this model needs all the information so we have opted to try training it with just one partition.



10.7. Neural Networks.

We defined a neural network model for binary classification with dropout layers to prevent overfitting. It sequentially trains the model on multiple data partitions, using each for one epoch per loop iteration. Early stopping is applied to halt training if the loss does not improve for three consecutive epochs, restoring the best weights afterward.



11. Adjusting the hyperparameters.

As the OPTUNA method worked previously, we decided to give it a go while adjusting the hyperparameters for our models. At first, it worked well, especially for linear regression. However, when we tried applying it to the Logistic Regression, the code took almost 700 minutes to execute just 10 out of 50 trials.

At that time we decided to change the method for the RandomisedSearchCV. This method worked relatively well for almost all of the models. However, when it came to KNN, the code took a long time to compute, so, just like in the case of defining our model, we decided to only use a small sample of data. We focused on df_train_1, which has around 59000 rows. At first, we tested hyperparameters with 1000 rows, and that worked well. 2000 and 10000 rows also seemed to work with no problems. We then tried to use 10% of the data, but it was impossible. After all, we decided to continue with the results we got for 10000 rows.

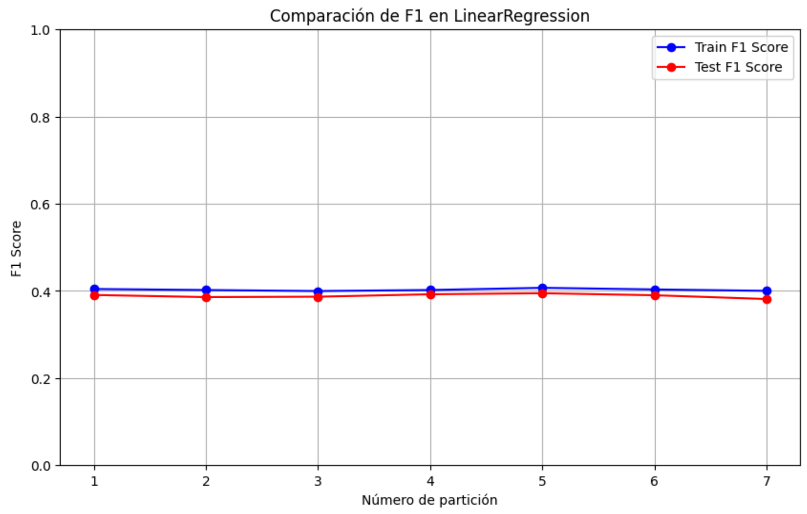
Finally, we achieved the following results:

- **LinearRegression**: {'alpha': 0.037554011884736255, 'eta0': 0.9517143064099162, 'penalty': 'elasticnet'}
- **PolynomialRegression**: {'alpha': 0.037554011884736255, 'eta0': 0.9517143064099162, 'penalty': 'elasticnet'}
- **LogisticRegression**: {'alpha': 0.09666320330745594, 'eta0': 0.8093973481164611, 'penalty': 'l2'}
- **DecisionTreeClassifier**: {'criterion': 'gini', 'max_depth': 4, 'max_features': 'log2', 'min_samples_leaf': 7, 'min_samples_split': 5}
- **KNN**: {'metric': 'manhattan', 'n_neighbors': 8, 'weights': 'distance'}
- **SVC**: {'alpha': 0.00694233026512157, 'max_iter': 3000, 'penalty': 'elasticnet', 'tol': 0.001}
- **Redes Neuronales**: {'neurons_1': 255, 'dropout_1': 0.31666966256912804, 'neurons_2': 124, 'dropout_2': 0.3008937986275955, 'optimizer': 'adam'}.

12. Training new models.

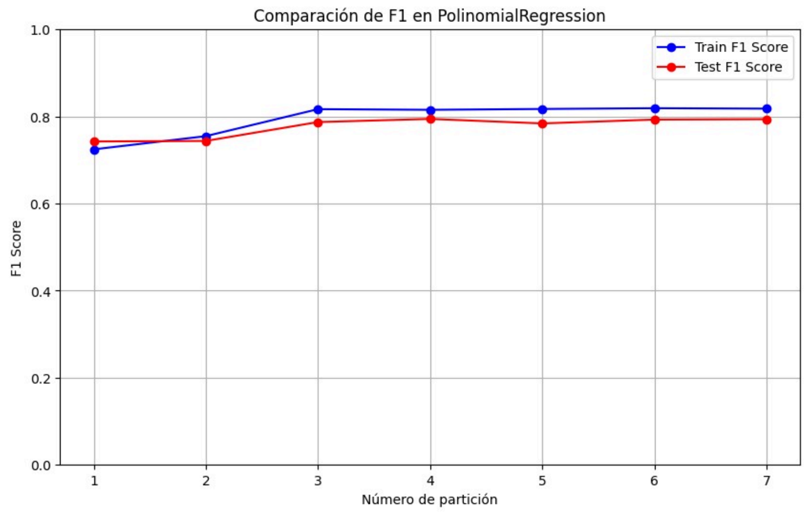
After finding the hyperparameters, we applied them to our models. Once again we used our functions to evaluate them and plot the results.

12.1. Linear Regression.



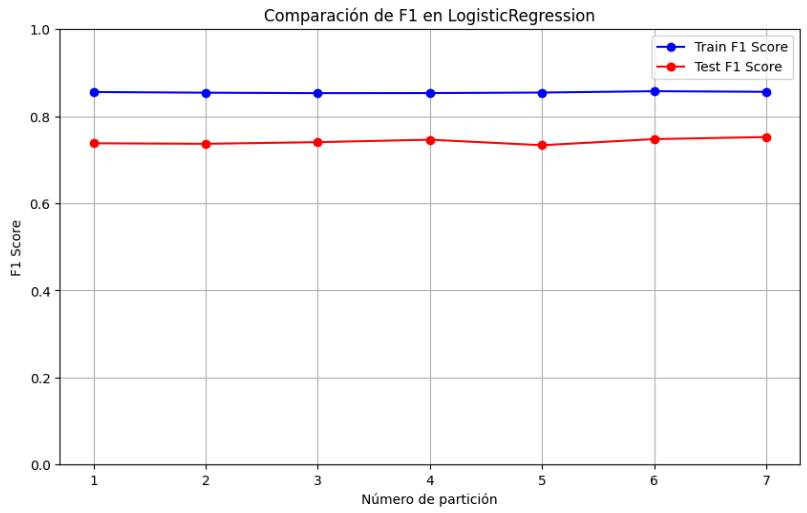
In the case of Linear Regression, the f1-score has definitely worsened. However, the hyperparameters helped with smoothing the results across the data frames.

12.2. Polynomial Regression.



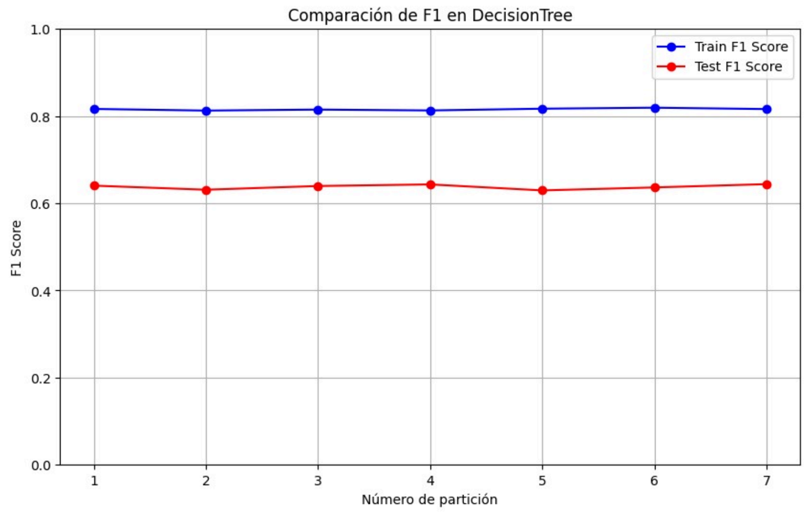
When it comes to Polynomial Regression, we can see a huge improvement in comparison with the previous model.

12.3. Logistic Regression.



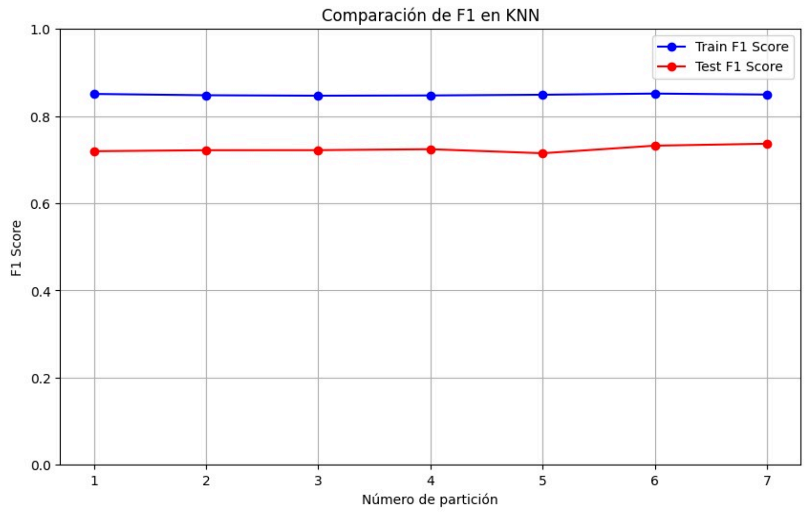
Logistic Regression has not visibly improved.

12.4. Decision Trees.



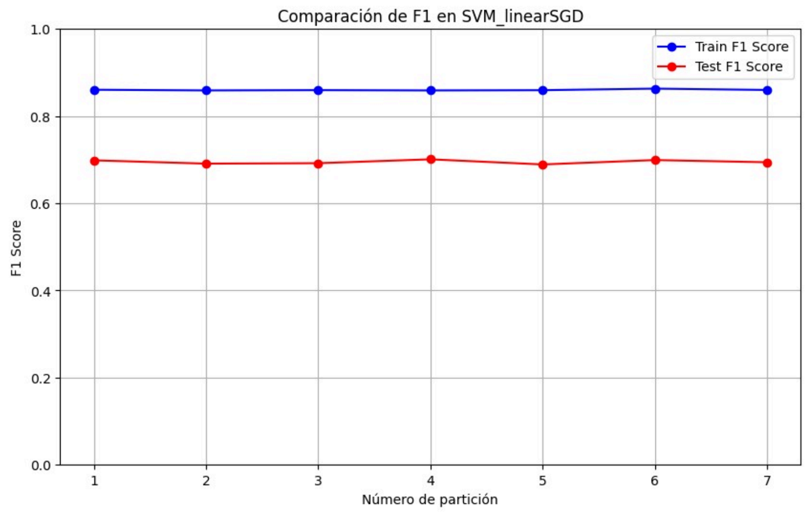
When it comes to Decision Trees, train data stayed the same, when the test worsened.

12.5. KNN.



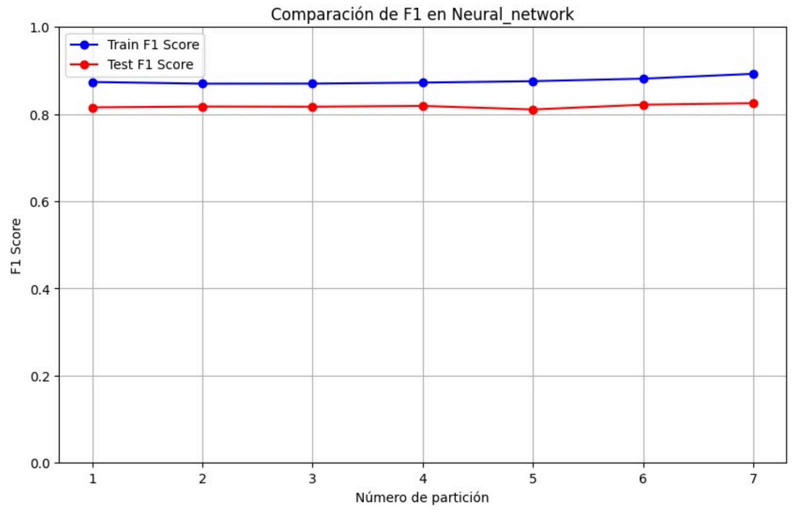
With the KNN, the results stayed similar.

12.6. SVM.



So did the SVM model.

12.7. Neural Networks.

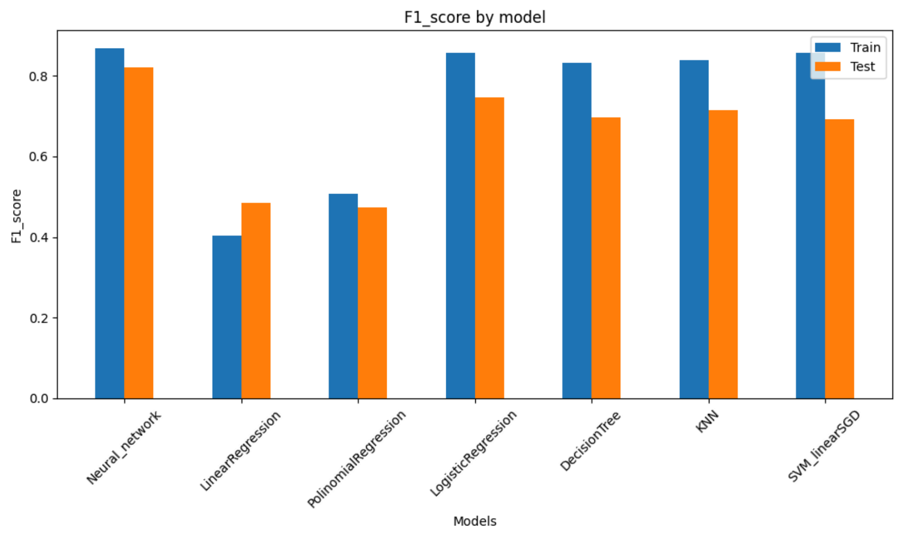


Neural Networks have improved a little.

13. Choosing the best model.

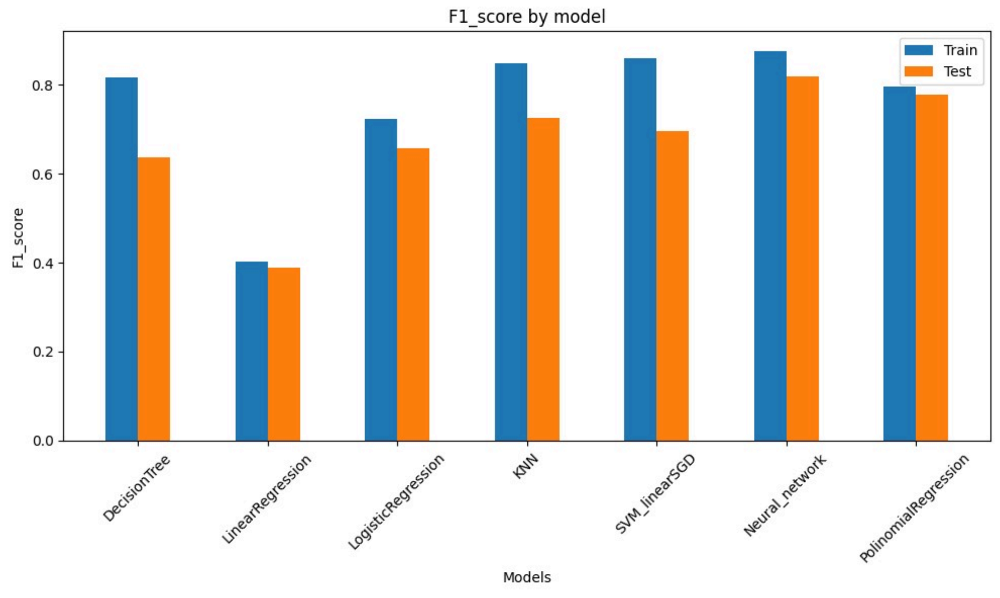
We compared the f1-score for all models before and after adjusting the hyperparameters.

13.1. Before adjusting the hyperparameters.



Judging from the results, we should probably choose Neutral Networks.

13.2. After adjusting the hyperparameters.



We can see that in most of the cases, there was not much of an improvement. Linear Regression worsened, whereas Polynomial Regression got relatively better. Other models stayed more or less the same.

As we have stated earlier, we will be using Neural Networks, adjusting the model with given hyperparameters: {'neurons_1': 255, 'dropout_1': 0.31666966256912804, 'neurons_2': 124, 'dropout_2': 0.3008937986275955, 'optimizer': 'adam'}.

14. Testing.

After choosing the best model, the only thing left to do was to test it on supplied data. We saved the result as *test_labels.csv*.

Apart from that, we supply the following:

- *preprocess.ipynb*
- *supervised.ipynb*
- *best_model.ipynb*
- *processed_dataset.csv.zip*
- and the given report, *memoria.pdf*.