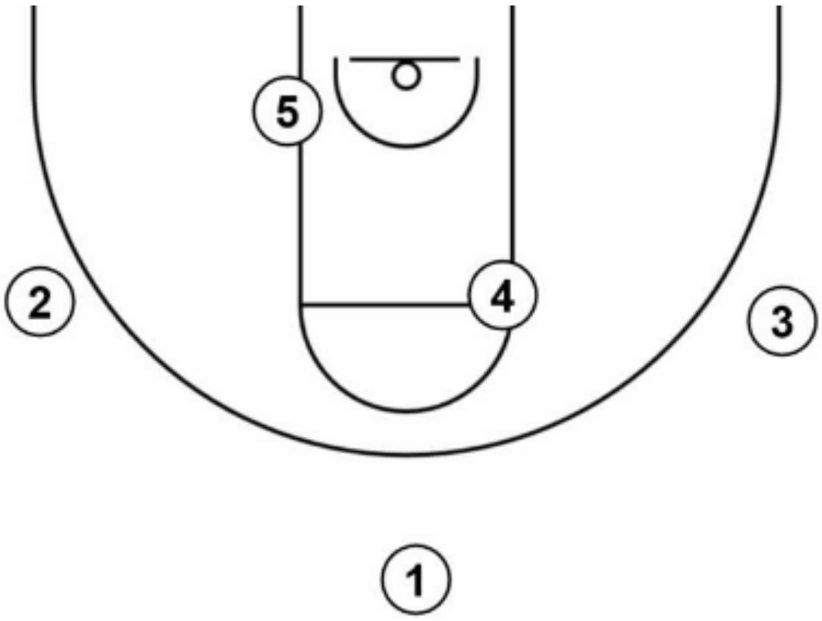
Universitat Politècnica de Catalunya

Facultad de Informática de Barcelona



NBA players position

Data mining Project 2

Members

Joan Areal Azuara

Laia Bonilla Pérez

Miguel Gutiérrez Jariod

Fujie Mei

Alessandro Scannavini

1. Description of the original data 5

Data collection process 5

Target of classification 5

Dataset 5

Missing data 6

2. Description of pre-processing of data 7

Preprocessing processes 7

Features section 10

Balancing dataset 12

Normalization 13

3. Evaluation criteria of data mining models 14

4. Execution of different machine learning methods 15

4.1 Naive-Bayes 15

Unbalanced dataset 15

Conclusion 16

4.2 K-NN 17

Simple Cross Validation 17

10-Fold Cross-Validation 18

Influence of the amount of data on the accuracy 19

Automatically find the best parameter 19

Apply model with best parameters found, trained with all training data to the test set 20

Mcnemar’s Test implementation 21

4.3 Decision Trees 22

4.4 Support vector machines 26

Linear model 26

Polynomial model 28

RBF kernel 29

4.5 Meta-learning algorithms 31

Voting Scheme 31

Bagging 34

Random Forest 35

Boosting 36

Conclusions about Meta-Learning algorithms 37

5. Comparison and conclusions 37

1. Description of the original data

Data collection process

Our dataset has been taken from Kaggle website and it's called 2021-2022 NBA Player Stats: 2021-2022 NBA Player Stats | Kaggle

This data collection contains two datasets, one corresponding to the statistics of the players in the NBA for the regular season of 2021-2022 and the other it's about playoffs statistics from the same NBA season. We have chosen to deal with the first dataset because it contains more players compared with the playoffs, so we can have more information to process.

Target of classification

The main objective of our project is to define the different positions that players can role in the game and classify it as being **close** or **away** from the basket respectively.

The role positions that players can be are the following:

Position	English	Spanish	Number Position
PG	Point Guard	Base	1
SG	Shooting Guard	Escolta	2
SF	Small Forward	Alero	3
PF	Power Forward	Ala-Pivot	4
C	Center	Pivot	5

In the game, the position structure of the players, drawn in the court, is a pentagon where the three superior vertices are the ones **further away** from the basket. These correspond to the **PG**, **SG** and **SF**. The remaining positions are the **closest** to the basket, **PF** and **C**.

Dataset

Our initial dataset is composed of 812 rows and 30 columns. Each of the rows corresponds to a player and the different columns contain information about his statistics. There are 3 categorical variables and 27 numerical variables. This is their description and meaning:

- **Rk** (type int) : Ranking ordering the players by their name.
- **Player** (type categorical) : Player's name.
- **Pos** (type categorical): Position of the player in the court.
- **Age** (type int) : Player's age.
- **Tm** (type categorical): Team where the player is joined.
- **G** (type int): Games played in the regular season.

- **GS** (type int): Games started in the regular season.
- **MP** (type float): Minutes played per game.
- **FG** (type float): Field goals per game.
- **FGA** (type float): Field goal attempts per game.
- **FG%** (type float) : Field goal percentage.
- **3P** (type float): 3-point field goals per game.
- **3PA** (type float) : 3-point field goal attempts per game.
- **3P%** (type float) : 3-point field goal percentage.
- **2P** (type float): 2-point field goals per game.
- **2PA** (type float): 2-point field goal attempts per game.
- **2P%** (type float) : 2-point field goal percentage.
- **eFG%** (type float) : Effective field goal percentage.
- **FT** (type float): Free throws per game.
- **FTA** (type float): Free throw attempts per game.
- **FT%** (type float) : Free throw percentage.
- **ORB** (type float): Offensive rebounds per game.
- **DRB** (type float): Defensive rebounds per game.
- **TRB** (type float): Total rebounds per game.
- **AST** (type float): Assists per game.
- **STL** (type float): Steals per game.
- **BLK** (type float): Blocks per game.
- **TOV** (type float): Turnovers per game, measures the number of times a team loses possession of the ball due to a mistake in a single game.
- **PF** (type float): Personal fouls per game.
- **PTS** (type float) : Points per game.

Missing data

To proceed with the preprocessing part, we have to look up for each of the variables trying to find nullable values and consider them to be treated. In our case we have seen that the dataset it's full of values, so we don't need to apply the mean or other algorithms to try to fill them with a corresponding value. We have *.isnull().any()* , which shows a table where all the variables appear and a boolean that indicates if there is any nullable.

2. Description of pre-processing of data

Preprocessing processes

Once we have some knowledge of the variables that compose our dataset and the relation with the missing values, we proceed to treat the dataset to use it in our models.

First of all, to begin with the reading of the dataset, we had to apply a type of encoding different to *UTF-8* because there were some problems with the reading of some values like players names. The one applied is *Windows-1252*.

The next step to follow was to see some descriptive information of each of the variables. For example, consider the mean, minimum and maximum for the numerical ones and see the range that was composed of each one. Moreover we take the information of the type on each variable, once the dataset was read, just to verify that it matches the numerical and categorical relation in the code.

Considering our target of the study being the classification of the positions of the players in a far or close range to the basket, proceed to treat our dataset. First of all, we look up for the variable *Pos*, which matches the position of each player.

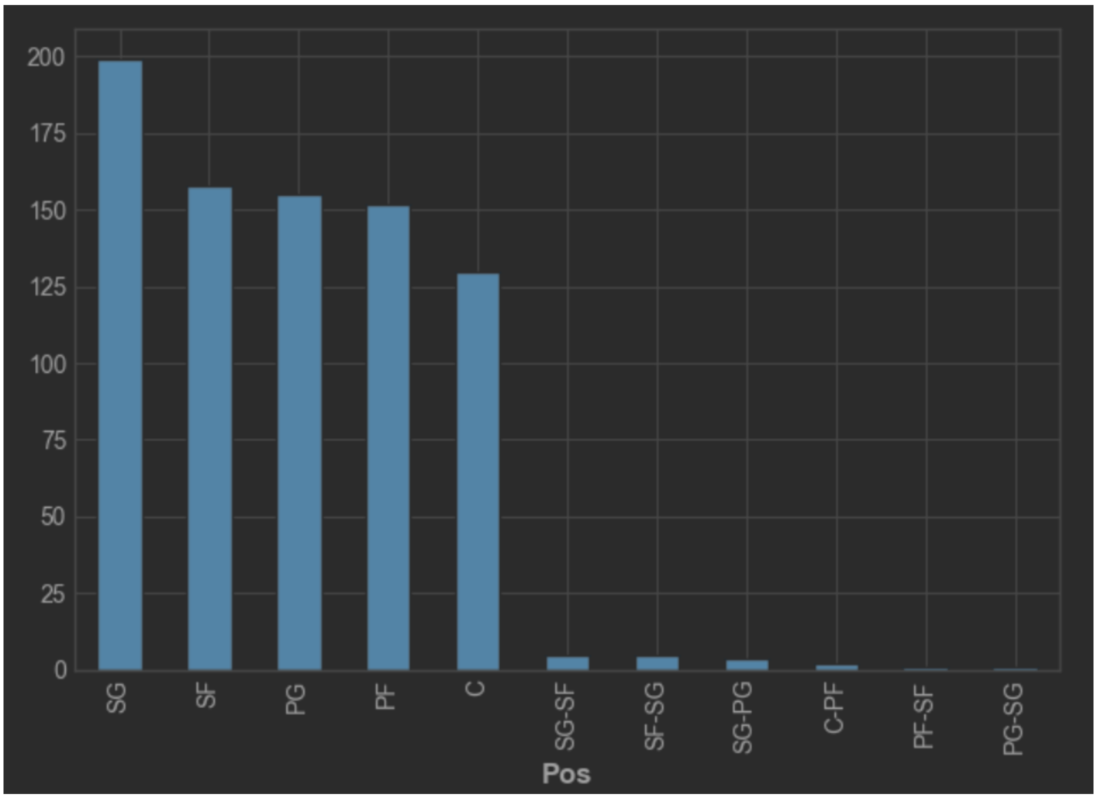
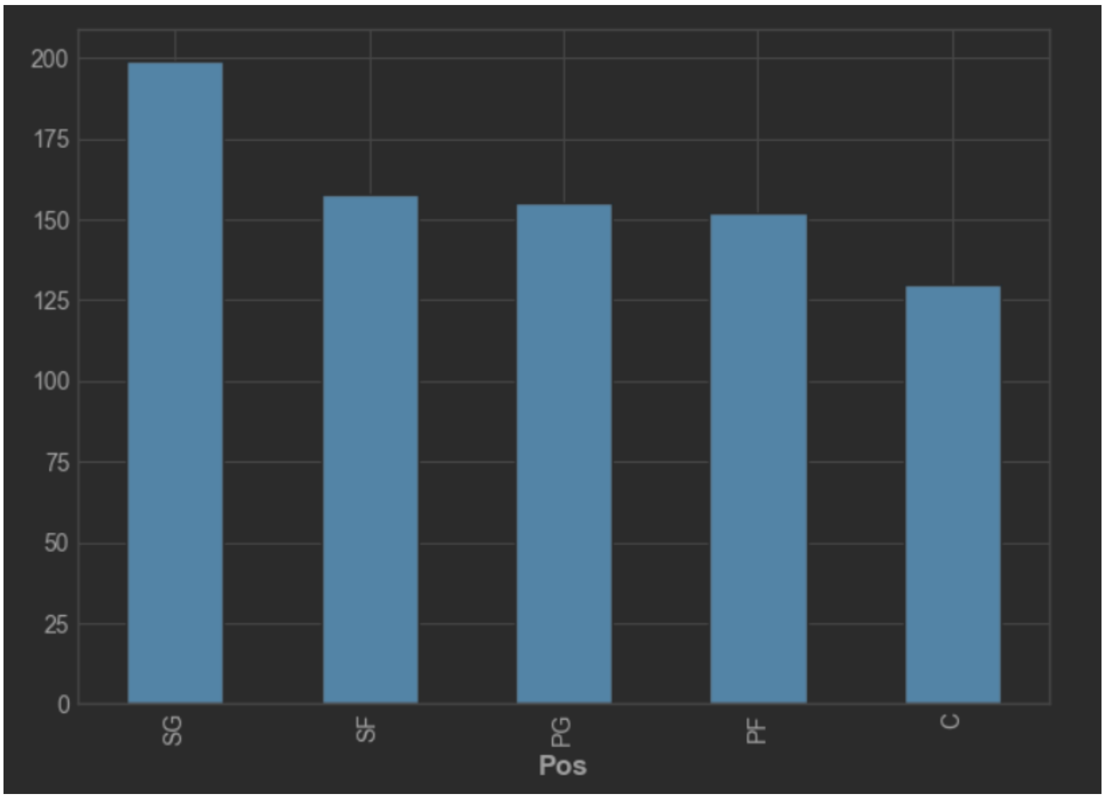


Fig: Number of players of each position

In the plot form above, we can appreciate each of the positions of players in the game. As our target is focused only in the main positions, not on the different roles that a player can play in the court, we have removed the possibility of having more than one role. Now the position attributes are the following:

Fig: Main positions in the court



To see the difference of the change, we can focus on the original and the after distribution:

Original distribution:

- 'Position 1' == PG -> 19.09%
- 'Position 2' == SG -> 24.51%
- 'Position 3' == SF -> 19.46%
- 'Position 4' == PF -> 18.72%
- 'Position 5' == C -> 16.01%

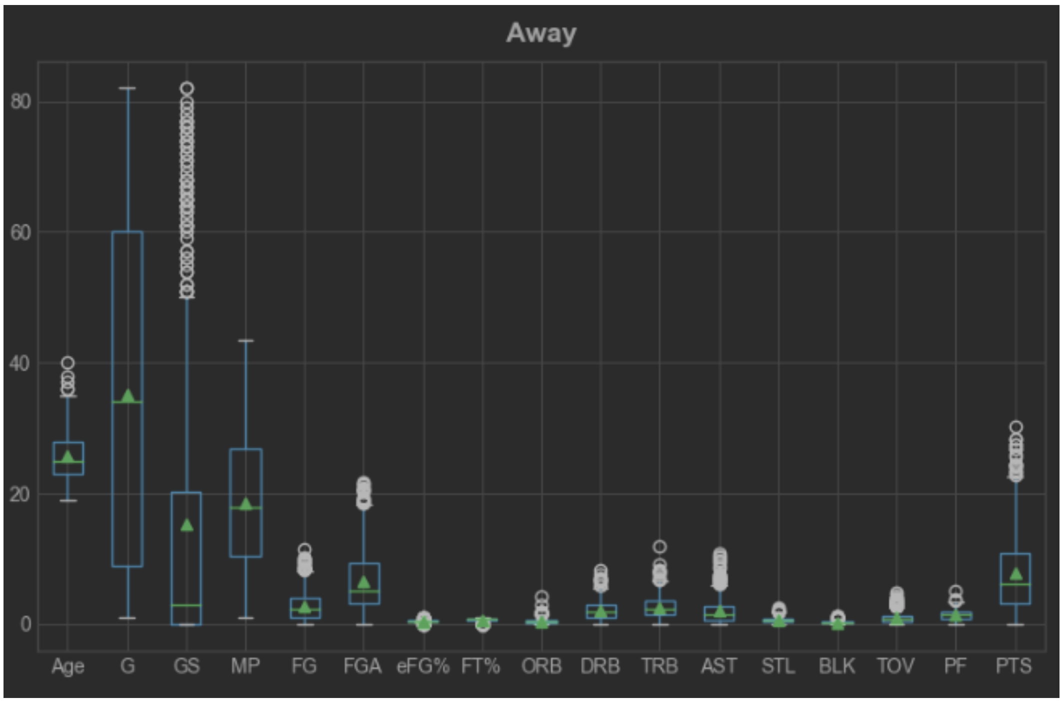
After sampling distribution:

- 'Position 1' == PG -> 19.52%
- 'Position 2' == SG -> 25.06%
- 'Position 3' == SF -> 19.9%
- 'Position 4' == PF -> 19.14%
- 'Position 5' == C -> 16.37%

There is not much difference between both distributions, an increase of the percentage in the second one it's the main point because of removing some of the players that have more than one role. Now we have more defined the main positions in the game for each player.

Considering our target, we defined a new column in the dataset related to the positions. So now, each player will be classified according to the proximity of the position in relation to the basket. As we describe in the target section, now we take the *away* or *close* attribute. Here we show the boxplots in relation to both parameters:

*Fig: variables in relation to away proximity paramete*r



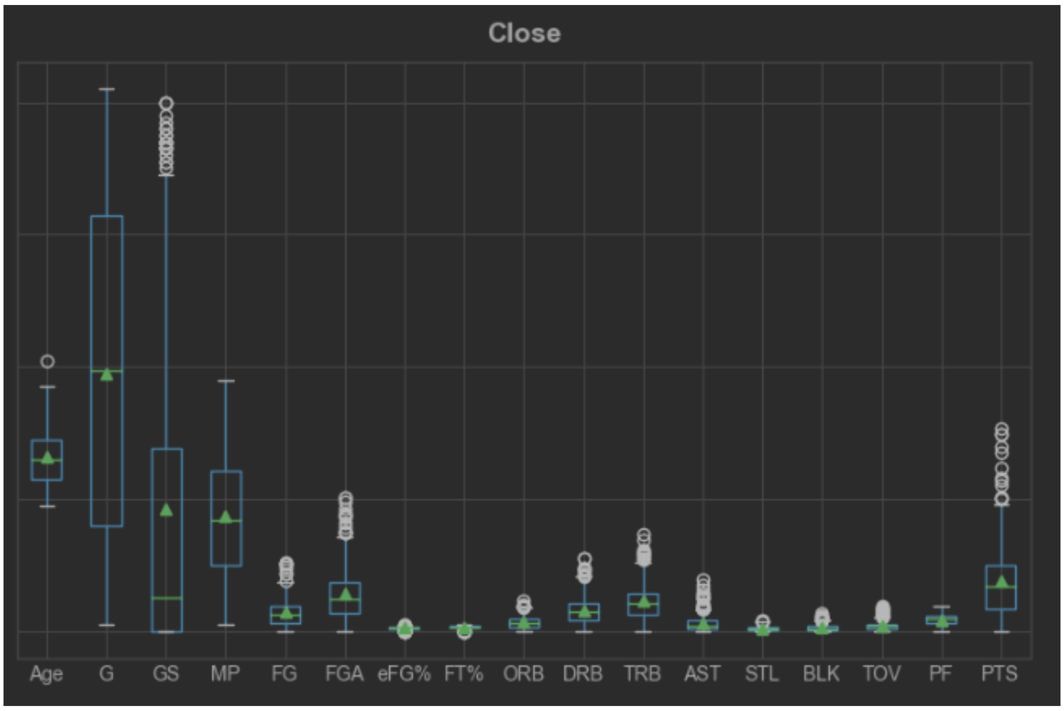


*Fig: variables in relation to close proximity paramete*r

Using methods *groupby()* and *describe()* we have obtained relevant information about the relationship between the parameters and the variables. Here we show up some of the most relevant in relation to the feature selection that we will show in the next section:

Away (512 players)	Close (282 players)
FG%: - mean: 0.393 - std: 0.140 **eFG%:** - mean: 0.466 - std: 0.158 **3PA:** - max: 11.7 - mean: 3.066 **2P%:** - mean: 0.461 - std: 0.187 **ORB:** - max: 4.3 - mean: 0.509 **DRB:** - max: 8.3 - mean: 2.080 **TRB:** - max: 12.0 - mean: 2.587 **BLK:** - max: 1.3 - mean: 0.240	**FG%:** - mean: 0.486 - std: 0.149 **eFG%:** - mean: 0.527 - std: 0.150 **3PA:** - max: 6.8 - mean: 1.579 **2P%:** - mean: 0.537 - std: 0.160 **ORB:** - max: 4.6 - mean: 1.373 **DRB:** - max: 11.0 - mean: 3.277 **TRB:** - max: 14.7 - mean: 4.651 **BLK:** - max: 2.8 - mean: 0.559

In general, we can see that a player, being in a closer position from the basket, used to take more blocks and rebounds, while the ones being away made more shots from the three line. Moreover we can comment that the ones closer have a little bit more effectiveness in the field goals by looking at the percentage of the two throws, this increase may be possible by the reduced distance of the shot from the hoop.

Features section

In this part, we consider treating our data for selecting the more relevant variables. To do this, we had a look at the correlations between variables and then applied the gain methodology.

By looking at the correlation, we have created a heat map in the preprocessing script. It allows us to identify in a faster way the ones more correlated, by having an orange color cell. As a summary, we can see that the most correlated values are the following:

Top	Absolute	Correlations
FG	PTS	0.990

FT	FTA	0.984
2P	2PA	0.979
FGA	PTS	0.976
DRB	TRB	0.972
FG	FGA	0.971
3P	3PA	0.970
FG%	eFG%	0.954
FG	2PA	0.945

Having a general view, we can determine that the most correlated ones are those that have a relation of a certain action and its number of attempts.

Once we have a main idea of the most correlated variables we decide to look up for a threshold, which can be applied in the gain method and take the most representative variables. We have applied the *sklearn* library, in which we have used functions like *SelectKBest()*, *mutual_info_classif()* and *cross_val_score()*. We obtained the following plot of the threshold selection:

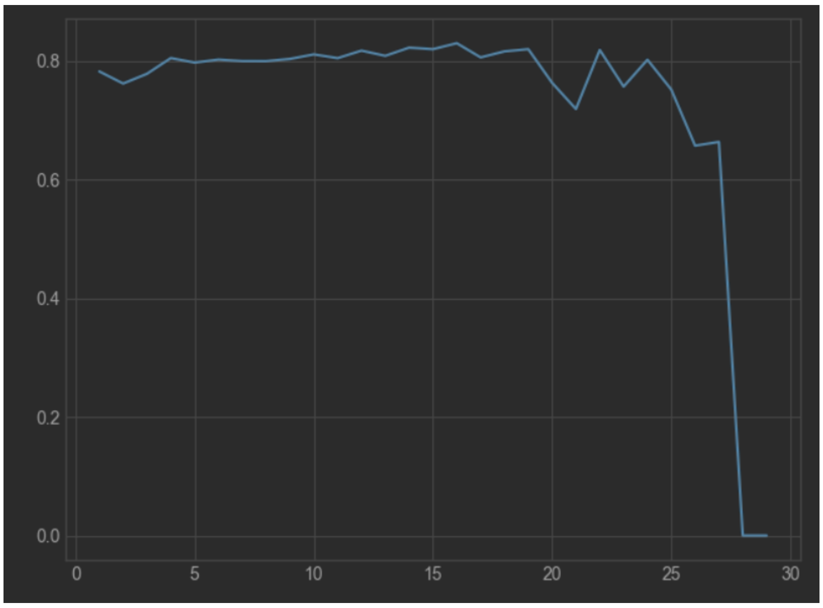


Fig: threshold selection

As we can see in the plot, the estimated threshold to apply should stay between 0.6 - 0.8 values. We think that it could be possible to use with correlation. In the next part we are going to see which one we consider properly for the gain methodology.

In this step, we apply the Gain method. To proceed with it, we had to transform the categorical variables into numerical ones. We used the label encoding, so the proximity variable changed its values to 0 or 1. We proceed to use the *mutual_info_classif()* from the *sklearn* library, which returns an array with the estimated mutual information between each feature and the target. We have used the value of 0.05 for the threshold, by looking at the variables of the following and considering with some knowledge in the game statistics, that was the best option.

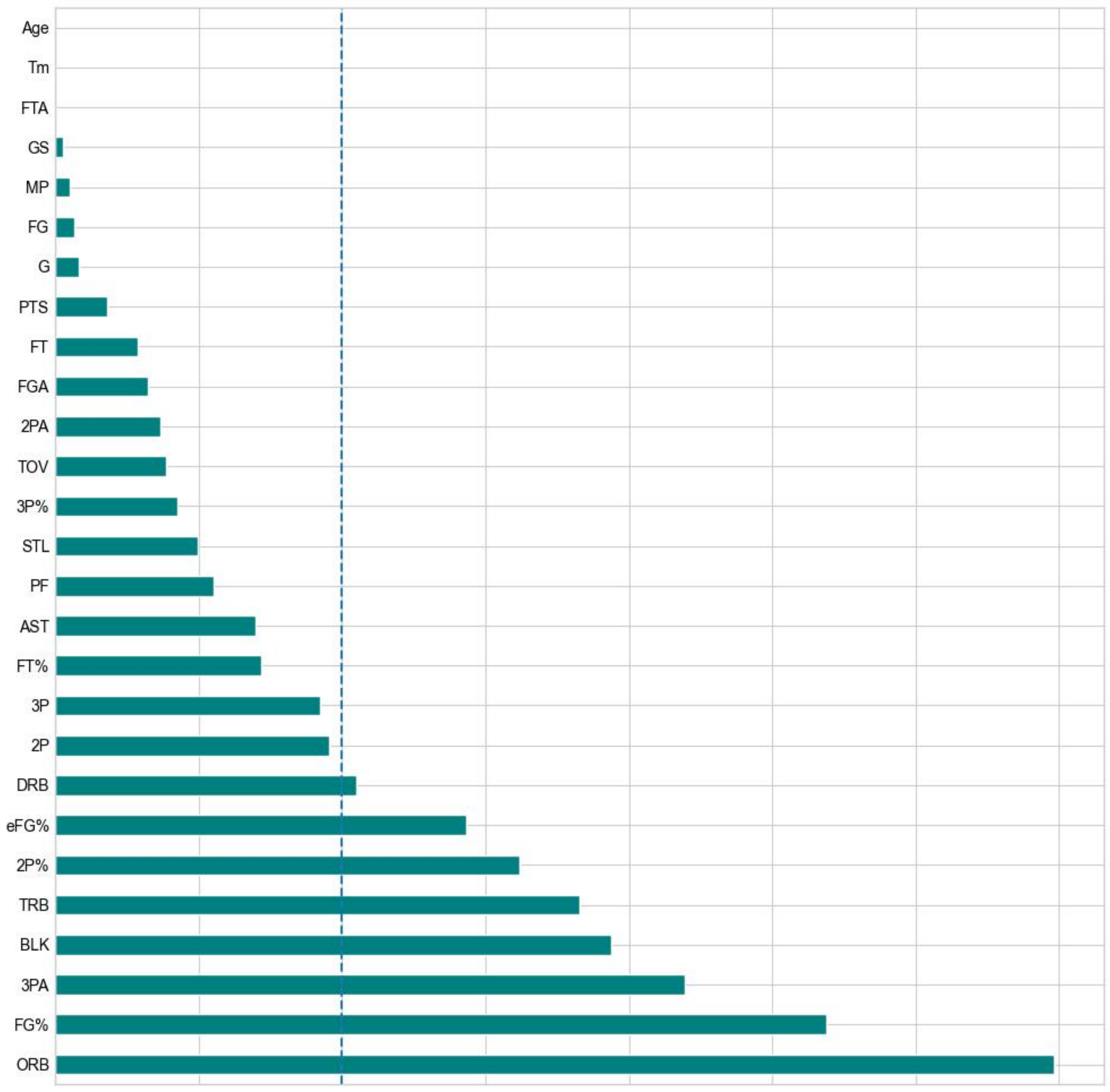


Fig: features selection

Finally the selected features, as shown in the image, are: ORB, FG%, 3PA, BLK, TRB, 2P%, eFG%, DRB.

Balancing dataset

In this part, we have seen that our dataset it's a little bit unbalanced. We have used the *RandomUnderSampler()* function from the *imblearn* library. Here are the results, where the main difference it's clearly shown.

Before balancing -> 794 rows in total

- 'Proximity' == 0 -> 35.52%
- 'Proximity' == 1 -> 64.48%

After balancing -> 564 rows in total

- 'Proximity' == 0 -> 50.0%
- 'Proximity' == 1 -> 50.0%

Normalization

In this section, once we have all the procedures explained before, we have normalized our dataset to use it in some models which should represent a better performance. So we have two datasets: unbalanced normalized and normal unbalanced one. As consider it in class, we are going to use only a dataset, just to maintain the same structure for all the models and see the authentic performance rather than use the most accurate dataset for the corresponding models.

3. Evaluation criteria of data mining models

To proceed with the evaluation of our models, we have chosen to split the 70% of our data set for training (555,8 elements) and the 30% remaining for the test one (238,2 elements). We also were thinking of applying 80/20, but finally we decided to apply the previous one, so the test evaluation contains a little bit more data to be considered.

In the majority of the cases that we have used cross validation, we have tried 10 fold cross validation. The reason to choose it with moderate values, usually used 10, it's because it reduces the variance while increasing the bias.

The principal metric used for the evaluation, being decided by the group, we are going to use the accuracy of the models after being trained to determine their effectiveness. We have to consider that our main dataset is unbalanced with a greater percentage of the “away” proximity positions, so we have to be carefully watching out that the good accuracy wasn’t from the over-representation of this one.

In addition we have preproceed our dataset making it balanced. In that way we can see whether or not balanced we obtain a significant boost in the accuracy of the models. That point is just for curiosity because in the end we are treating different datasets, so we decide to apply the unbalanced one to obtain the models results.

4. Execution of different machine learning methods

4.1 Naive-Bayes

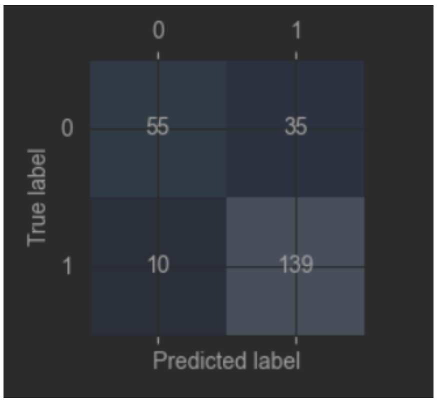
To perform this method, we decided to make predictions of the dataset taken from the preprocessing part. We have realized two predictions, one by fit (on the testing set) and the other by cross validation (to the entire data).

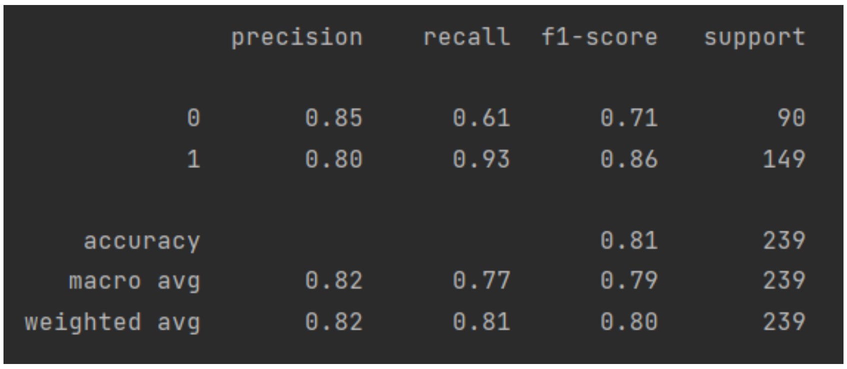
Unbalanced dataset

Accuracy: 0.8117154811715481

Confusion matrix: *Precision, recall, F-measure for fit method:*

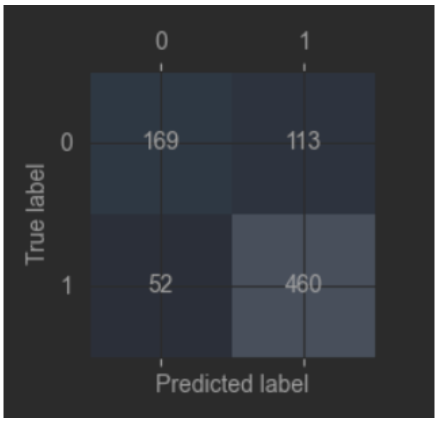
Interval of confidence: (0.7621523926656273, 0.861278569677469)

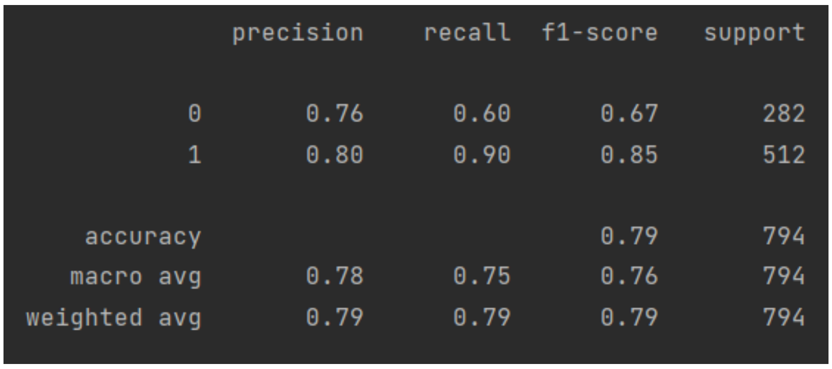



```
              precision    recall  f1-score   support

           0       0.85      0.61      0.71        90
           1       0.80      0.93      0.86       149

    accuracy                           0.81       239
   macro avg       0.82      0.77      0.79       239
weighted avg       0.82      0.81      0.80       239
```

Accuracy: 0.792191435768262

Confusion matrix: *Precision, recall, F-measure for cross validation method:*




```
              precision    recall  f1-score   support

           0       0.76      0.60      0.67       282
           1       0.80      0.90      0.85       512

    accuracy                           0.79       794
   macro avg       0.78      0.75      0.76       794
weighted avg       0.79      0.79      0.79       794
```

Interval of confidence: (0.7639696408112039, 0.8204132307253201)

In this case, we can appreciate that the fit method is based on the 30% of the dataset, while the cross one focuses on all the data. On both, we can see as being unbalanced, that the “away” position is higher than the “close” position. For the first methodology we have obtained a 0.81 of accuracy while in the other a 0.79. We think that it is possible due to the fit methodology, because it doesn’t take randomly data from the dataset, as the cross does, so it's possible that has take more relevance for the close one, but as the figure shows there are more of the away ones, so it’s kind of confusing. We believe that having applied the cross validation to an unbalanced dataset will give us the best performance. As both performances show, the accuracy is practically the same.

Conclusion

In conclusion, for our dataset, the use of the Navie Bayes method has performed good results. We consider that all the performances have given good samples of accuracy. We believe that the performance is fine because the majority of the attributes are independent, few of them are related. We can consider independent taking into account that is not the same as throwing 2P than 3P shoots, as an example, because as they are related in the game, by the act of shooting, at the end are two different kinds of values for the statistics of the players.

We have observed that both classes are well predicted, having 76% for the closest and 80% for the aways one in precision. It's reliable that having 35% of the data versus 64% gives us this precision.

Finally, applying the use of fit or cross validation shows us similar results, there is not so much difference.

4.2 K-NN

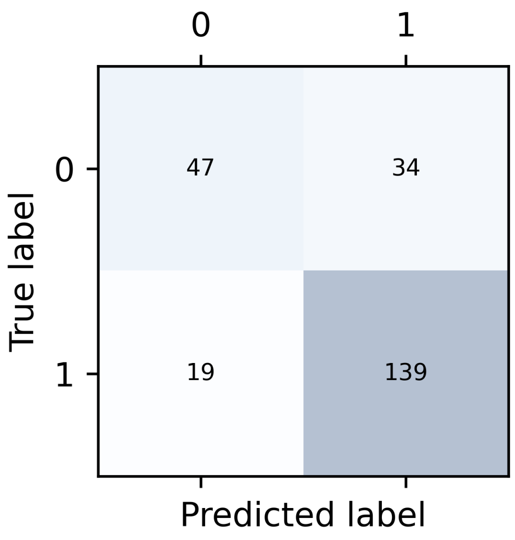
In order to keep the results of the various algorithms comparable, even if a normalized dataset performs slightly better with K-NN, we decided to maintain a non normalized one.

In order to train the classifier, a division in the dataset was applied, the proportion chosen is 70% for training and 30% for validation.

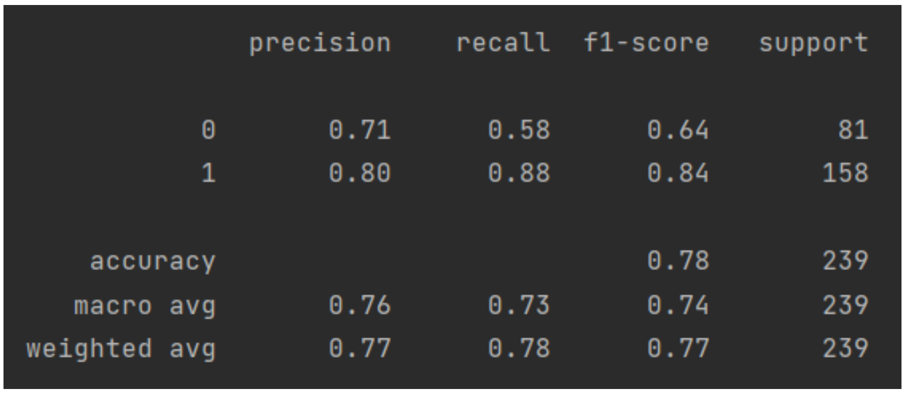
Simple Cross Validation

Applying the simple cross validation we achieved the following results:

- Accuracy: 0.7782
- Confusion matrix:



- Precision, recall, F-measure:



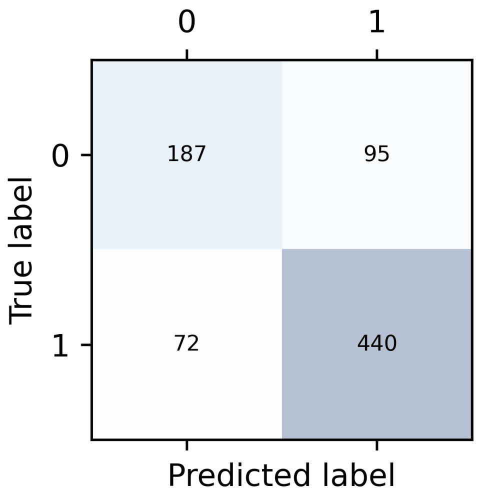
```
              precision    recall  f1-score   support

           0       0.71      0.58      0.64        81
           1       0.80      0.88      0.84       158

    accuracy                           0.78       239
   macro avg       0.76      0.73      0.74       239
weighted avg       0.77      0.78      0.77       239
```

The result of the precision of both the classes are quite satisfactory, since we were able to reach a value greater than 0.70 in both cases. For what concerns the recall instead the value of the class ‘0’ is significantly smaller (0.58) than the other (0.88) which means that we are really good in identifying individuals from class ‘1’ while some trouble arises when we have to find a member of the class ‘0’. This is probably due to the fact that the second class is more represented in the dataset.

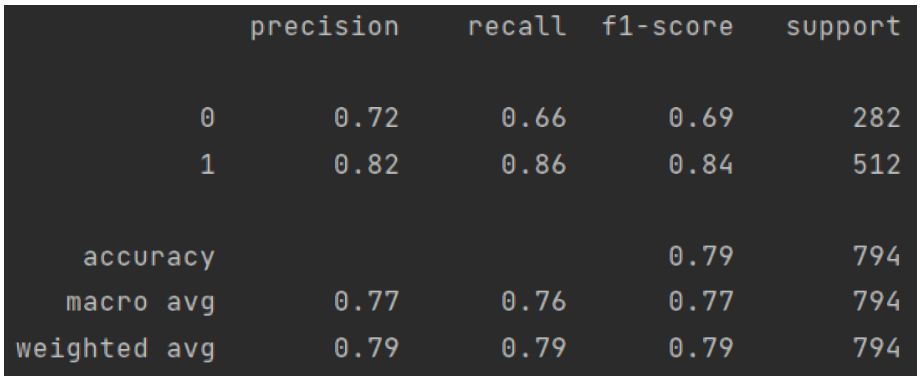
The same happens for the f1 score, even in this case the results of the class ‘1’ are better, but also for class ‘0’ are satisfactory.

10-Fold Cross-Validation

- Mean accuracy: 0.7897
- Standard deviation: 0.0341
- Confusion matrix:



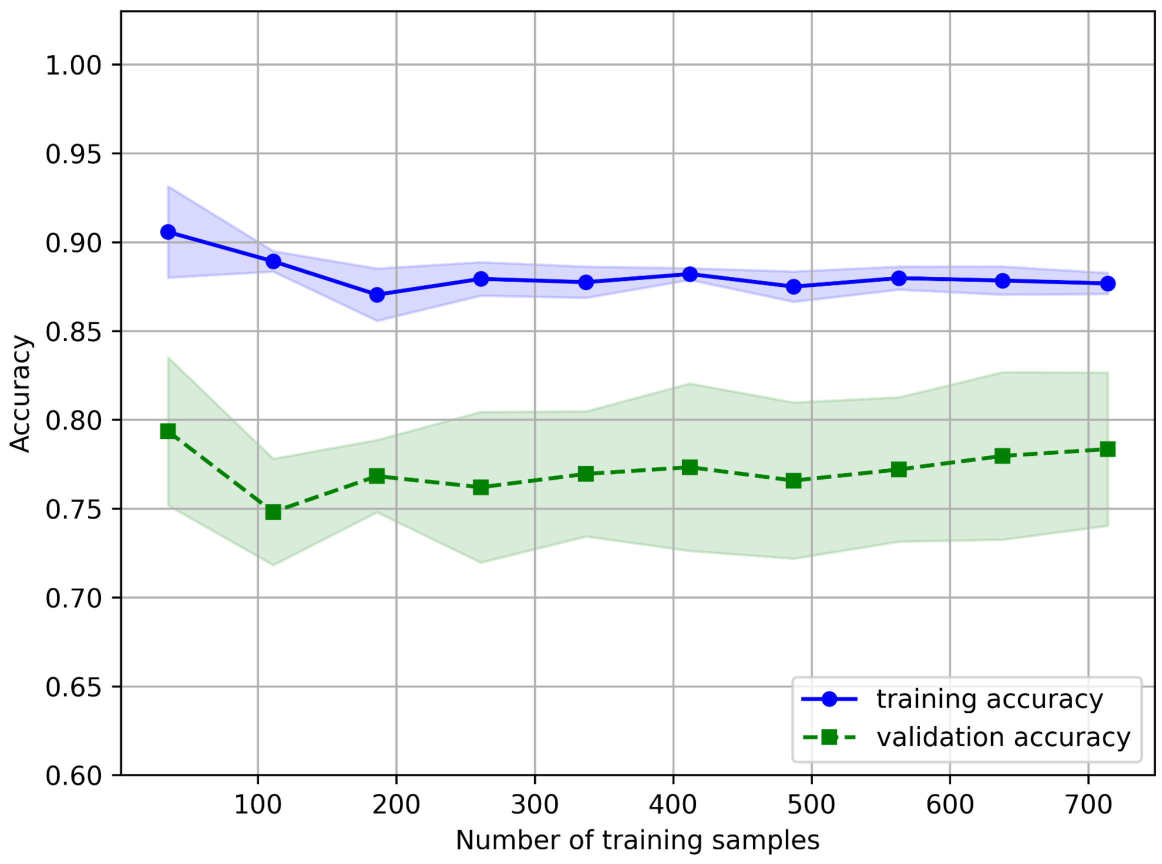
- Precision, recall, F-measure:



```
              precision    recall  f1-score   support

           0       0.72      0.66      0.69       282
           1       0.82      0.86      0.84       512

    accuracy                           0.79       794
   macro avg       0.77      0.76      0.77       794
weighted avg       0.79      0.79      0.79       794
```

With 10-Fold Cross-Validation we get better results on both the classes for basically every measure. The only one slightly decreasing is the recall for class ‘1’, which decreases by 2%, but it comes with an improvement of 8% on the other class. Also in this case the overall accuracy is quite high, and the standard deviation gives goods results as well.

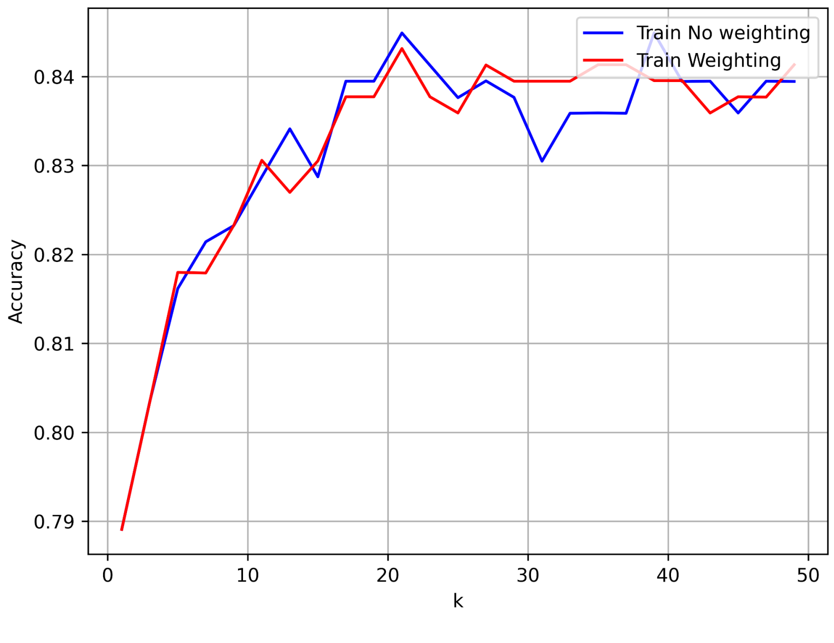
Influence of the amount of data on the accuracy



As it's clearly visible in the plot above, the amount of data influences the accuracy only for the first 200 training samples, after that amount the accuracy tends to stabilize around 0.88 for the training and around 0.78 for the validation.

Automatically find the best parameter

- Graph showing the different accuracy varying K, both weighted and not weighted training are shown:



As it’s clearly visible from the graph the best value in the accuracy is reached for K equals to 21, there is no significant difference between weighting and not weighting even if the second one performs slightly better.

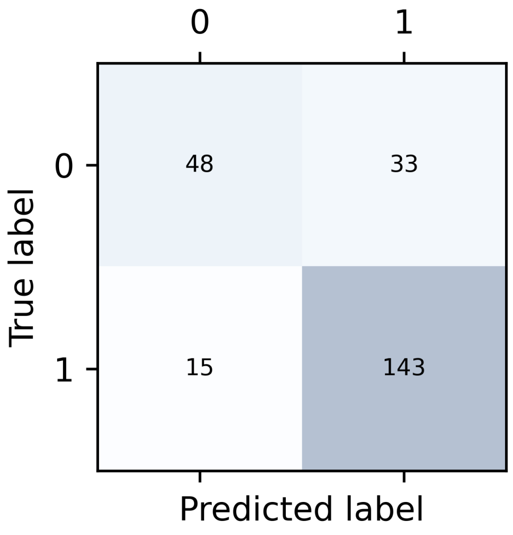
- Grid Search: for sake of completeness we decided also to use the Grid Search method in python for searching the best value for K and we got almost the same results:
  - Nr of neighbors: 21
  - Not weighted
  - Accuracy: 0.8449

Apply model with best parameters found, trained with all training data to the test set

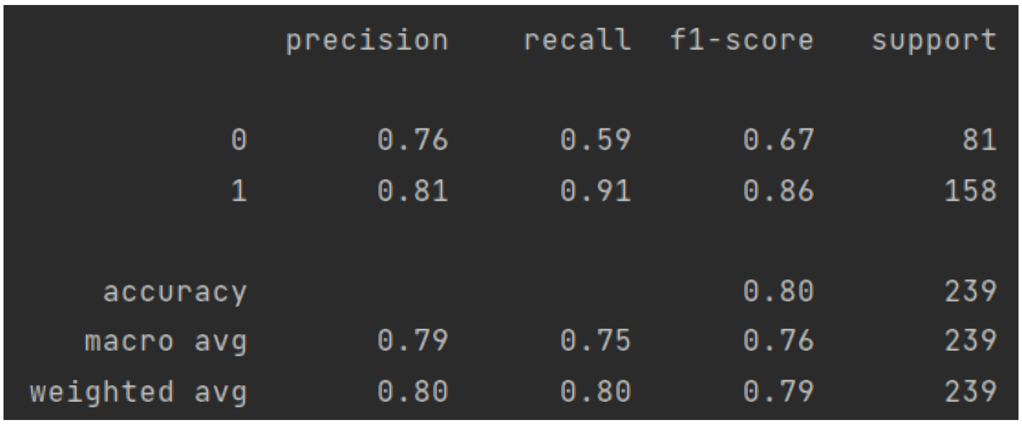
We are now going to apply the model with the best parameters found above, which are number of neighbors equals to 21 and uniform weights:

- Accuracy: 0.7991

- Confusion matrix:



- Precision, recall, F-measure:



```
              precision    recall  f1-score   support

           0       0.76      0.59      0.67        81
           1       0.81      0.91      0.86       158

    accuracy                           0.80       239
   macro avg       0.79      0.75      0.76       239
weighted avg       0.80      0.80      0.79       239
```

As can be seen from the report above, applying the best parameters gives us best results on every single measure, even though the improvement is quite small.

Mcnemar's Test implementation

As we know from the theory, having a large value of K can be useful since it smooths the decisions frontiers and makes the model less sensitive to noise, but at the same time as K increases we also lose locality in the decision, we increase the computational time and the KNN algorithm fits a smoother curve to the data. This is due to the fact that a higher value of K reduces the edginess by taking more data into account, reducing the overall complexity and flexibility of the model. For these reasons we decided to test smaller values of K in order to understand if there were actually a big difference between the best model found (K = 21) and other models with smaller values of K. In order to do that we decided to use the Mcnemar's Test, which is a non-parametric test that applies to 2 x 2 frequency tables generally used to verify the existence of differences in dichotomic data (positive/negative) before and after a certain change or treatment.

In order to apply the test we first we had to built the following table:

	Number of elements correctly predicted from the second model	Number of element wrongly predicted from the second model
Number of elements correctly predicted from the first model	a	b
Number of elements wrongly predicted from the first model	c	d

and then we used the following formula checking if X^2 was greater or smaller than 3.84.

$$\chi^2 = \frac{(|b-c|-1)^2}{(b+c)}$$

Since applying the test with K = 7 (smallest value with good results) and K = 21 (best parameter) we end up with $X^2 < 3.84$ we can state that there is no significant difference between the two models' accuracies and due to that also K = 7 is a good parameter. This result could be useful if some changes in the amount of data will be done (increasing the dataset could increase the computational time with high K) and also to have a model which reflects different types of goals (reducing/increasing flexibility of the model).

4.3 Decision Trees

Before doing anything else, we have chosen to do this with the unbalanced dataset. We trained our model and checked its accuracy with the test data. Initially we didn't specify the min_samples_split and the min_impurity_decrease, and we obtained the following results:

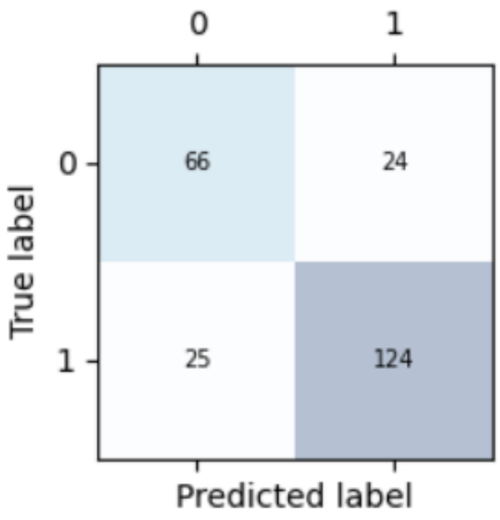
```
Accuracy: 0.7949790794979079
```

Class Name	precision	recall	f1-score	support
0	0.73	0.73	0.73	90
1	0.84	0.83	0.84	149

macro avg	0.78	0.78	239	None
weighted avg	0.80	0.79	0.80	239
	None	None	None	None

```
(0.7437960391452927, 0.8461621198505231)
```

And a confusion matrix that looked like this:



We then repeated the same but specifying a min_samples_split and a min_impurity_decrease of 2 and 0.02 respectively. We chose these values after trying with different ones since we considered they made the tree more readable and gave more accurate results. This time the values we got looked like this:

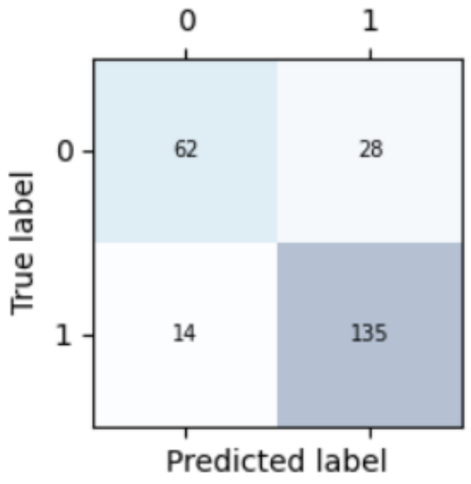
```
Accuracy: 0.8242677824267782
```

Class Name	precision	recall	f1-score	support
0	0.82	0.69	0.75	90
1	0.83	0.91	0.87	149

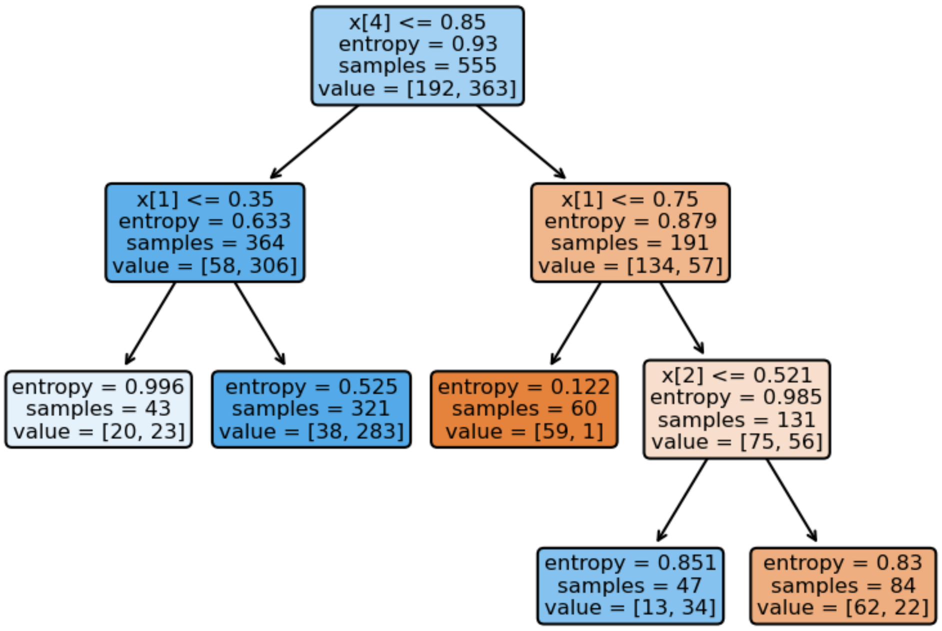
macro avg	0.82	0.81	239	None
weighted avg	0.82	0.82	0.82	239
	None	None	None	None

```
Interval of confidence: (0.7760164847424902, 0.8725190801110663)
```

With this the accuracy was 3% higher than the previous times, but repeating the execution multiple times we observed that these two accuracies had some variance and sometimes the first version ended up having a higher one than the second one, so this difference was not so meaningful. This is the confusion matrix we got, also quite similar:

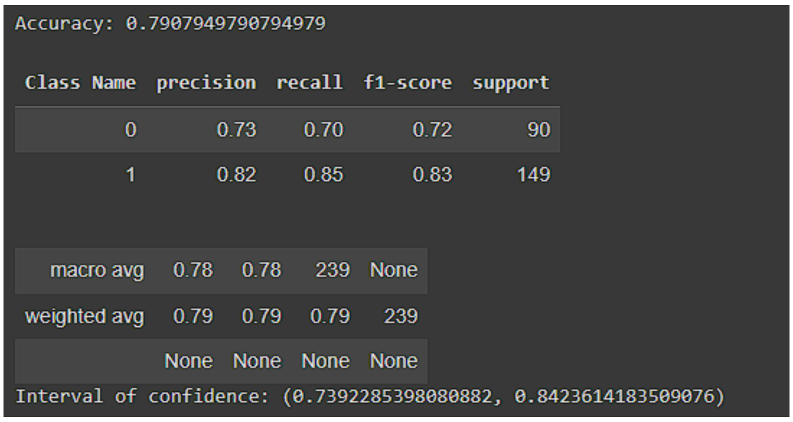


We obtained the representation of the decision tree we trained in this second version:



From this decision tree we saw that DRB, 3PA and 2P% were the most discriminating attributes.The entropy values we obtained are pretty high, which is not a good sign.

We did one last test, to obtain the gini index measures. The Gini index measures the probability of misclassifying a random chosen element from the set, so if it is lower the node will be more pure and more likely to give correct information. We trained a decision tree to see the gini index on each node, for this tree we got this accuracy and confusion matrix:

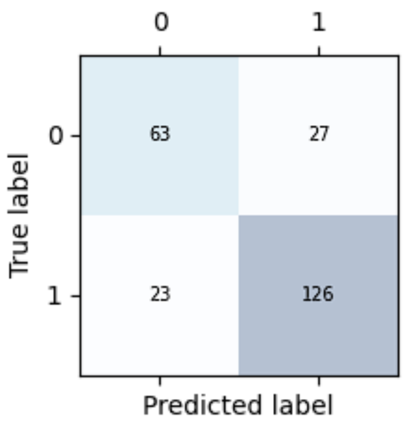


```
Accuracy: 0.7907949790794979
```

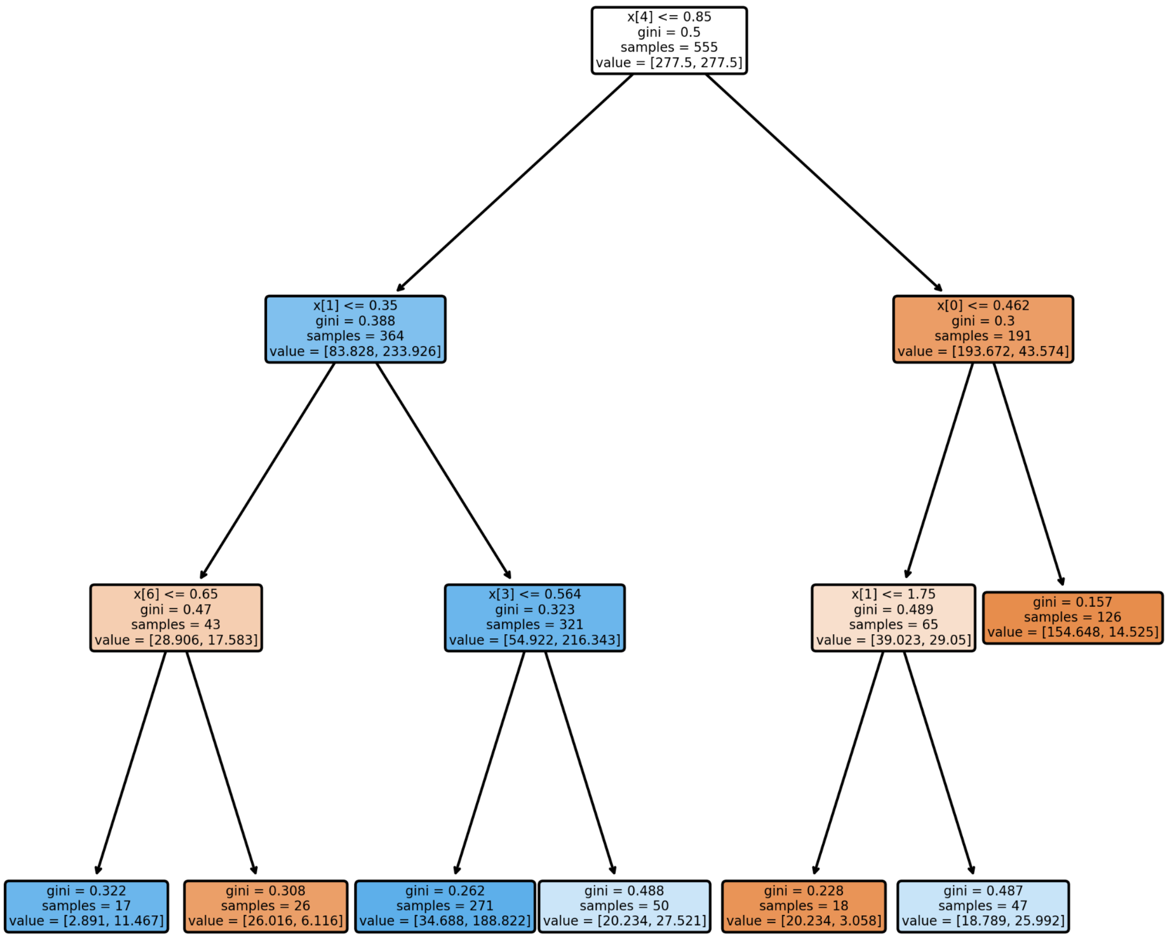
Class Name	precision	recall	f1-score	support
0	0.73	0.70	0.72	90
1	0.82	0.85	0.83	149

macro avg	0.78	0.78	239	None
weighted avg	0.79	0.79	0.79	239
	None	None	None	None

```
Interval of confidence: (0.7392285398080882, 0.8423614183509076)
```



$$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 63 & 27 \\ 1 & 23 & 126 \end{array}$$



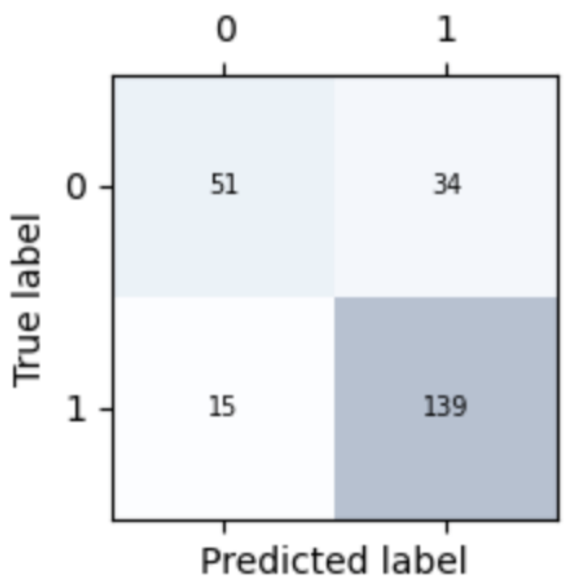
From this tree we get the information that X[0] which is FG%, and X[3] which is eFG% have the purest gini index. These two have a considerably high amount of samples, X[3] more than X[0], this together with the fact that they have a pretty low impurity index makes them valuable nodes. X[1] is also similar to those two on the left side of the tree, but on the right side it has a worse gini value. X[6] has a worse gini value and less samples than the rest, so it’s not as valuable.

4.4 Support vector machines

In this method we executed 3 types of kernels on our dataset: Linear kernel, polynomial kernel and RBF kernel. This is the method where we had to wait the longest to finish the execution even when our number of samples isn't that high, but the algorithm has a cubic cost relative to the number of rows. We didn't reduce the number of columns since there weren't many, since they've been withdrawn in the preprocessing step as well as we kept the same number of rows since it's limited, so in order to speed up the algorithm there wasn't anything we could do.
The execution time of the GridSearch took the longest in the polynomial model, with 8h, so we only experimented with 2 degrees.

Linear model

First we tried an SVM with default parameters. As we can see below the results obtained aren't that bad with a 79% of accuracy and acceptable values on recall and precision. As we can see in the confusion matrix, the prediction from this model are acceptable but with a few errors.

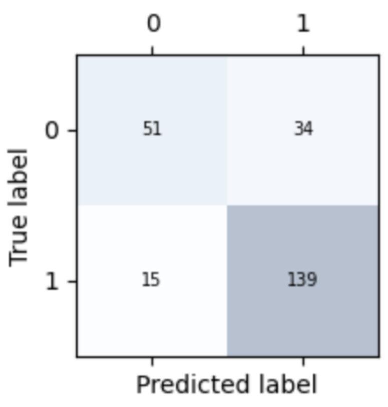


```
Accuracy on test set:  0.7949790794979079
```

Class Name	precision	recall	f1-score	support
0	0.77	0.60	0.68	85
1	0.80	0.90	0.85	154

macro avg	0.79	0.76	239	None
weighted avg	0.79	0.79	0.79	239

But we can do better than this,we have a regularization parameter C which has to be adjusted and we used GridSearch for this.
As we can see, the best one we got is with C=10, but even when adjusting for this C we can't really see an improvement, it even stays the same.
We can also see that the proportions of supports falls into the high extreme of the acceptable range, with a 41%. The acceptable range of supports goes from 20%-40%.



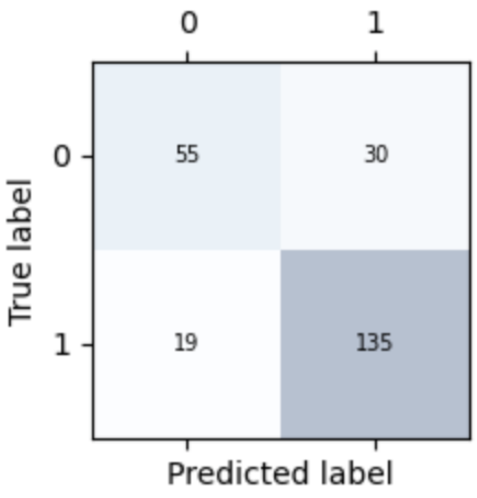
```
Accuracy on test set:  0.7949790794979079

Best value of parameter C found:  {'C': 10.0}

Number of supports:  227 ( 219 of them have slacks)
Prop. of supports:  0.409009009009009
```

Class Name	precision	recall	f1-score	support
0	0.77	0.60	0.68	85
1	0.80	0.90	0.85	154

macro avg	0.79	0.76	239	None
weighted avg	0.79	0.79	0.79	239

Polynomial model

This polynomial kernel has been performed only with a quadratic polynomial kernel (degree = 2). Unfortunately, we could not get results from a cubic polynomial kernel (degree = 3), as it never finished the execution after more than 10 hours running.
As we can see in the pictures below, the results are identical to the linear model with almost the same accuracy confusion matrix.



```
Accuracy on test set:  0.7949790794979079
```

Class Name	precision	recall	f1-score	support
0	0.74	0.65	0.69	85
1	0.82	0.88	0.85	154

macro avg	0.78	0.77	239	None
weighted avg	0.79	0.79	0.79	239

But this time when tuning C using GridSearch we can see some improvements in terms of accuracy, but very slightly. But the proportions of supports goes up which isn’t a good thing, in this case we can also see improvements in the confusion matrix, but it’s miniscule.

```
Confusion matrix on test set:
 [[ 55  30]
 [ 17 137]]

Accuracy on test set:  0.803347280334728

Best combination of parameters found:  {'C': 10.0}

Number of supports:  236 ( 214 of them have slacks)
Prop. of supports:  0.4252252252252252
```

RBF kernel

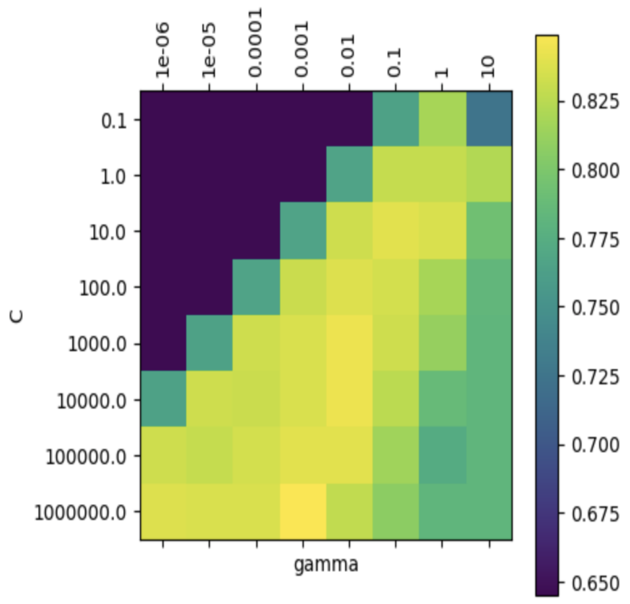
We repeat the process but with RBF kernel, the results obtained are almost the same in the other 2 models, we haven't seen great improvement which would tell us to use this instead of the others. A first run gives us the following results, where we can see that the accuracy goes up 82,4% and confusion matrix is also slightly better but not by much.

```
Confusion matrix on test set:
 [[ 55  30]
 [ 12 142]]

Accuracy on test set:  0.8242677824267782
```

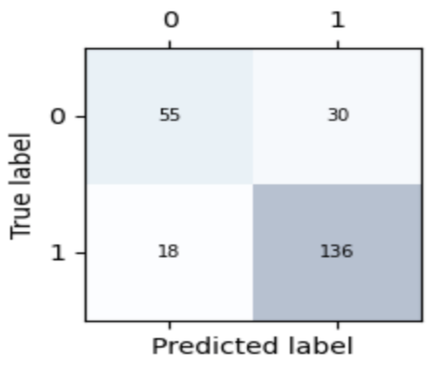
Class Name	precision	recall	f1-score	support
0	0.82	0.65	0.72	85
1	0.83	0.92	0.87	154

macro avg	0.82	0.80	239	None
weighted avg	0.82	0.82	0.82	239

But it gets weird when we fine-tune it to the best parameters, because the accuracy even goes down by a little bit. But in this model we get the best proportions of supports vectors with a 38%, which perfectly fits in the ideal range.



```
Best combination of parameters found:  {'C': 1000000.0, 'gamma': 0.001}
```



```
Accuracy on test set:  0.799163179916318

Best value of parameter C found:  {'C': 1000000.0, 'gamma': 0.001}

Number of supports:  213 ( 185 of them have slacks)
Prop. of supports:  0.3837837837837838
```

Class Name	precision	recall	f1-score	support
0	0.75	0.65	0.70	85
1	0.82	0.88	0.85	154

macro avg	0.79	0.77	239	None
weighted avg	0.80	0.80	0.80	239

4.5 Meta-learning algorithms

We have performed 4 different Meta-learning methods: Voting Scheme, Bagging, Random Forest and Boosting. We have done all with the unbalanced dataset.

Voting Scheme

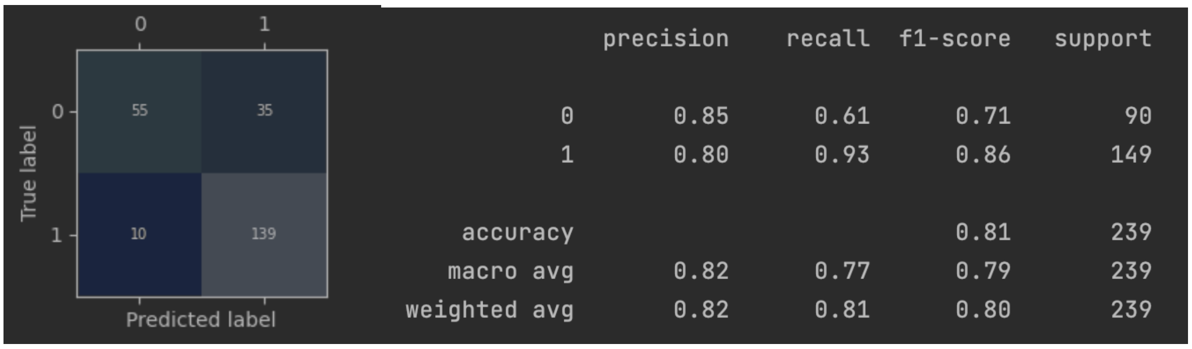
To vote for the class for each individual, we have used three classifiers: Naive Bayes, KNN and Decision Tree. We have applied this method twice, one with majority voting and the other with weighted voting.

First of all, we have made a train for each method with each classifier. As in the previous methods, the division of the dataset is 70% for training and 30% for validation. We have obtained the following results:

<u>Naive Bayes</u>

Accuracy: 0.8117154811715481

Confusion matrix: *Precision, recall, F-measure:*

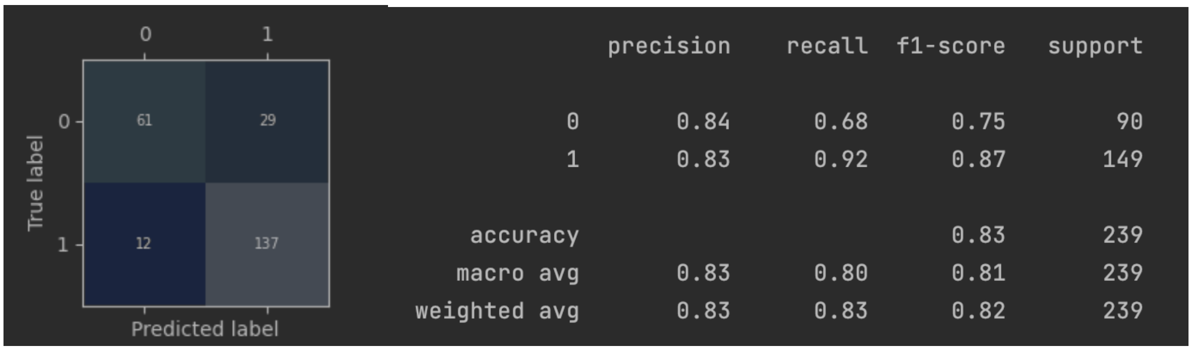


```
              precision    recall  f1-score   support

           0       0.85      0.61      0.71        90
           1       0.80      0.93      0.86       149

    accuracy                           0.81       239
   macro avg       0.82      0.77      0.79       239
weighted avg       0.82      0.81      0.80       239
```

<u>KNN</u>

Accuracy: 0.8284518828451883

Confusion matrix: *Precision, recall, F-measure:*

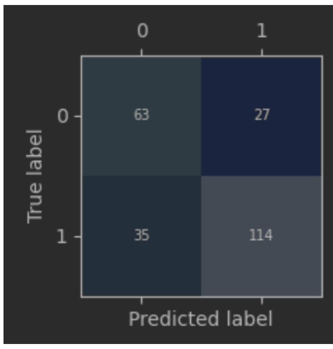


```
              precision    recall  f1-score   support

           0       0.84      0.68      0.75        90
           1       0.83      0.92      0.87       149

    accuracy                           0.83       239
   macro avg       0.83      0.80      0.81       239
weighted avg       0.83      0.83      0.82       239
```

<u>Decision Tree</u>

Accuracy: 0.7405857740585774

Confusion matrix: *Precision, recall, F-measure:*



```
              precision    recall  f1-score   support

           0       0.64      0.70      0.67        90
           1       0.81      0.77      0.79       149

    accuracy                           0.74       239
   macro avg       0.73      0.73      0.73       239
weighted avg       0.75      0.74      0.74       239
```

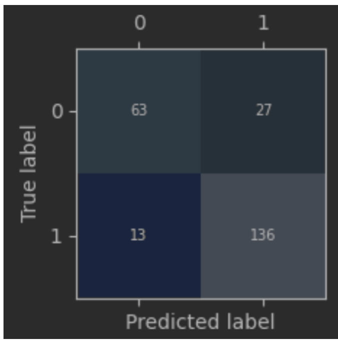
The two firsts classifiers give good results, around 80% of accuracy. Although the one that gives us better results is the KNN, with an accuracy of 0.83. The worst classifier is the Decision tree with an accuracy of 0.74.

- **Majority Voting**

We have applied the majority voting, and we have obtained the following results:

Accuracy: 0.8326359832635983

Confusion matrix: *Precision, recall, F-measure:*



```
              precision    recall  f1-score   support

           0       0.83      0.70      0.76        90
           1       0.83      0.91      0.87       149

    accuracy                           0.83       239
   macro avg       0.83      0.81      0.82       239
weighted avg       0.83      0.83      0.83       239
```

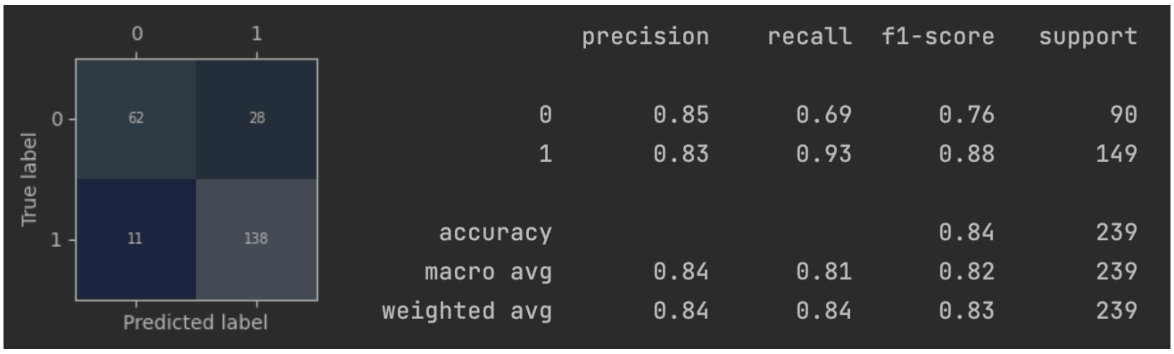
We can see that both classes have the same precision, and an accuracy of 0.83.

- **Weighted Voting**

We know that KNN is the best classifier, Naive Bayes the second, and Decision Tree is the worst. So we have put weights of [3,2,1] respectively to do the voting.
With the weighted voting we have obtained better results than with the majority voting, because the classifiers with best accuracy have more weight in the voting.

Accuracy: 0.8368200836820083

Confusion matrix: *Precision, recall, F-measure:*



```
              precision    recall  f1-score   support

           0       0.85      0.69      0.76        90
           1       0.83      0.93      0.88       149

    accuracy                           0.84       239
   macro avg       0.84      0.81      0.82       239
weighted avg       0.84      0.84      0.83       239
```

As we can see, this type of voting scheme method gives an accuracy of 84%. The precision of both classes is good, but especially the “0” class precision (85%).

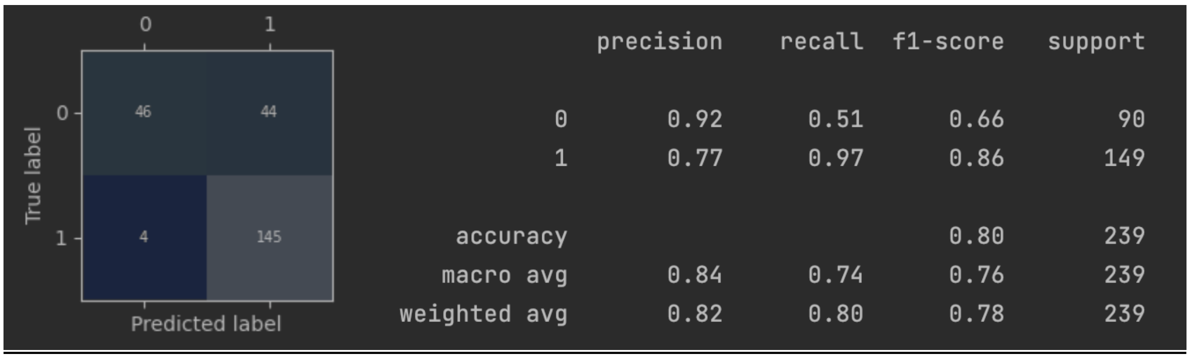
Bagging

To apply the Bagging method, we have done different baggings for each classifier (Naive Bayes, KNN and Decision Tree). We have fixed the parameters max_features to 0.35 and n_estimators to 200, because these values give us the best results.

<u>Naive Bayes</u>

Accuracy: 0.799163179916318

Confusion matrix: *Precision, recall, F-measure:*

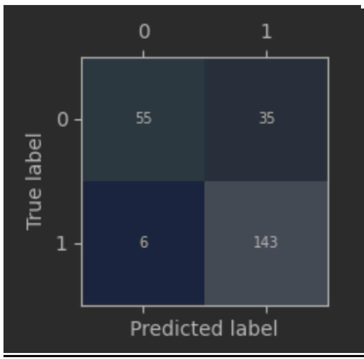


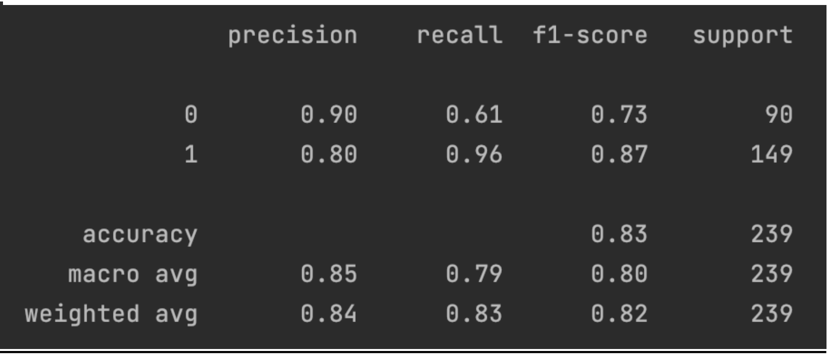
```
              precision    recall  f1-score   support

           0       0.92      0.51      0.66        90
           1       0.77      0.97      0.86       149

    accuracy                           0.80       239
   macro avg       0.84      0.74      0.76       239
weighted avg       0.82      0.80      0.78       239
```

<u>KNN</u>

Accuracy: 0.8284518828451883

Confusion matrix: *Precision, recall, F-measure:*



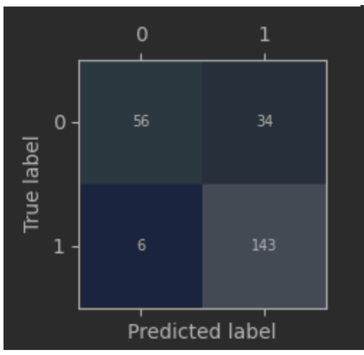


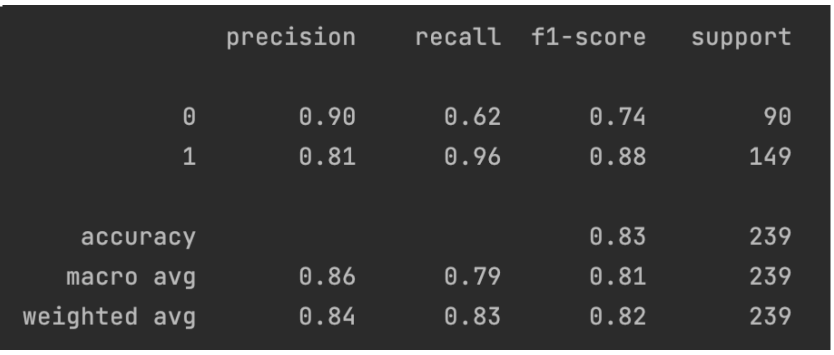
```
              precision    recall  f1-score   support

           0       0.90      0.61      0.73        90
           1       0.80      0.96      0.87       149

    accuracy                           0.83       239
   macro avg       0.85      0.79      0.80       239
weighted avg       0.84      0.83      0.82       239
```

Decision Tree

Accuracy: 0.8326359832635983

Confusion matrix: *Precision, recall, F-measure:*





```
              precision    recall  f1-score   support

           0       0.90      0.62      0.74        90
           1       0.81      0.96      0.88       149

    accuracy                           0.83       239
   macro avg       0.86      0.79      0.81       239
weighted avg       0.84      0.83      0.82       239
```

As we can see in the previous images, the bagging of the three classifiers give good results. The worst is Naive Bayes with an accuracy of 80%, and the other two (KNN and Decision Tree) have the same accuracy (83%). Although with the decimals we can appreciate that the Decision Tree classifier is the best.

We can observe that the classifiers predict the minority class (“0” class) better than the majority class (“1” class). In all cases the precision of the first is around 90%, while the precision of the second is around 80%.

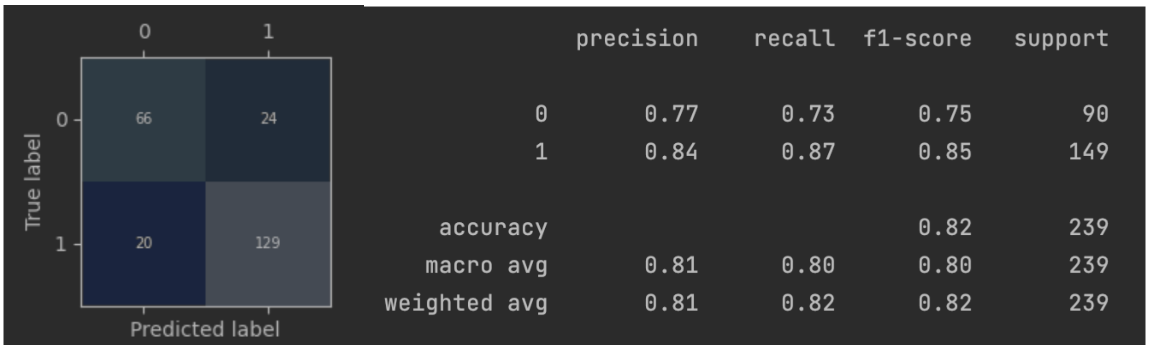
Random Forest

For the Random Forest method, we have decided to train two random forests. One with the max_features parameter not fixed and with n_estimators to 100, and the other with max_features fixed to 0.35 and with n_estimators to 200.

Random forest with n_estimators = 100

Accuracy: 0.8117154811715481

Confusion matrix: *Precision, recall, F-measure:*

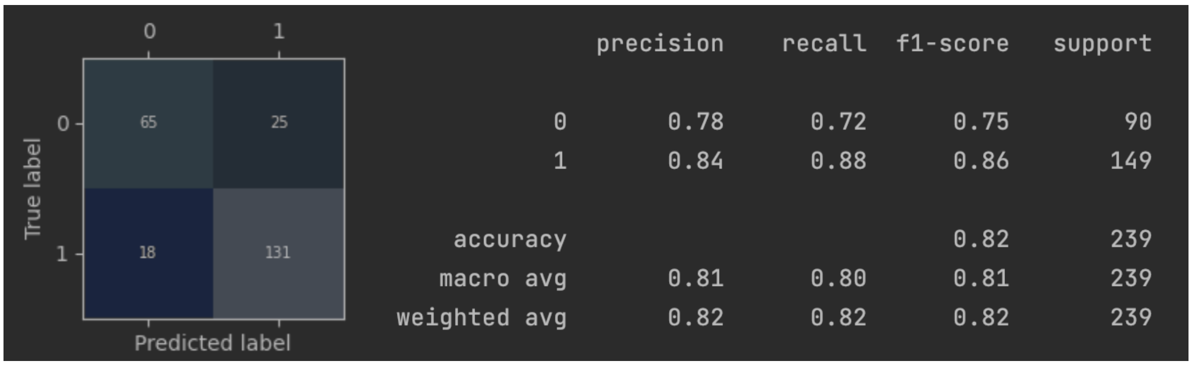


```
              precision    recall  f1-score   support

           0       0.77      0.73      0.75        90
           1       0.84      0.87      0.85       149

    accuracy                           0.82       239
   macro avg       0.81      0.80      0.80       239
weighted avg       0.81      0.82      0.82       239
```

<u>Random forest with n_estimators = 200 and max_features = 0.35</u>

Accuracy: 0.8200836820083682

Confusion matrix: *Precision, recall, F-measure:*



```
              precision    recall  f1-score   support

           0       0.78      0.72      0.75        90
           1       0.84      0.88      0.86       149

    accuracy                           0.82       239
   macro avg       0.81      0.80      0.81       239
weighted avg       0.82      0.82      0.82       239
```

We can observe that the results of the two random forests are practically identical, both with an accuracy of 82%. The classifier predicts a bit better the majority class (“1” class).

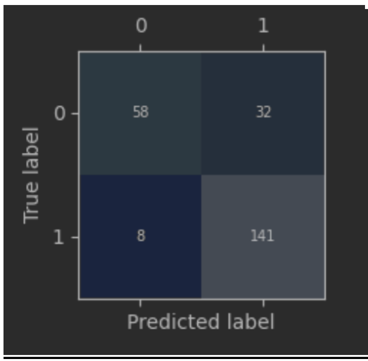
Boosting

We have performed tho methods of boosting: AdaBoost classifier and GradientBoosting classifier. We have put the value of n_estimators to 100 in AdaBoost and to 50 in GradientBoosting, because using them we obtain the highest accuracy.

<u>AdaBoostClassifier</u>

Accuracy: 0.8326359832635983

Confusion matrix: *Precision, recall, F-measure:*

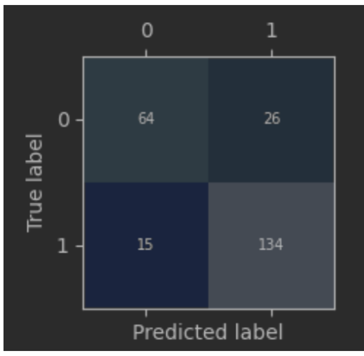


```
              precision    recall  f1-score   support

           0       0.88      0.64      0.74        90
           1       0.82      0.95      0.88       149

    accuracy                           0.83       239
   macro avg       0.85      0.80      0.81       239
weighted avg       0.84      0.83      0.83       239
```

<u>GradientBoostingClassifier</u>

Accuracy: 0.8284518828451883

Confusion matrix: *Precision, recall, F-measure:*



```
              precision    recall  f1-score   support

           0       0.81      0.71      0.76        90
           1       0.84      0.90      0.87       149

    accuracy                           0.83       239
   macro avg       0.82      0.81      0.81       239
weighted avg       0.83      0.83      0.83       239
```

With boosting we obtain an 83% of accuracy in both of the methods performed. We can observe a little difference between AdaBoost and GradientBoosting, which is that in the first method, the minority class is the best predicted, while in the second method, the class with higher precision is the majority class.

Conclusions about Meta-Learning algorithms

With all the meta-learning methods we have obtained good results. The best accuracy is obtained with the weighted voting (84%). Although there aren't bad methods, none have less than a 80% of accuracy.

5. Comparison and conclusions

Model			Accuracy	Interval of confidence
Naive Bayes	fit		**0.81**	[0.76, 0.86]
	cross validation		0.79	[0.76, 0.82]
KNN	simple cross		0.78	[0.73, 0.83]
	10 fold cross validation		0.79	[0.78, 0.83]
	Apply best model		**0.80**	[0.74, 0.84]
Decision Trees	entropy		0.75	[0.69, 0.80]
	entropy, min_samples_split = 2, min_impurity_decrease = 0.02		**0.82**	[0.77, 0.87]
	fit		0.79	[0.73, 0.84]
SVM	linear		0.79	[0.79, 0.82]
	polynomial		0.79	
	RBF		**0.82**	
Meta-learning	Voting Scheme	Majority	0.83	[0.78, 0.88]
		Weighted	**0.84**	[0.79, 0.88]
	Bagging	NB	0.80	[0.75, 0.85]
		KNN	0.83	[0.78, 0.88]
		Dec.Tree	0.83	[0.78, 0.88]
	Random Forests		0.81	[0.76, 0.86]
	Boosting	Ada Boost	0.83	[0.78, 0.88]
		Gradient Boosting	0.83	[0.78, 0.88]

After having applied all the data mining methods, we have obtained an accuracy around 0.80 in all of them. The best method, as we can see in the following table, is the meta-learning algorithm Voting Scheme with weighted voting, with a 84% of accuracy. The method that gives worse results is Decision Trees, with a 75% of accuracy, but is not a bad result.

As seeing that all models have a similar accuracy, we have difficulties choosing a main one and represent a clear difference between the behaviors. In general, the accuracy obtained on the validation dataset has been similar to that obtained with cross validation, usually the last one gives us a better performance.

Finally, we can conclude that choosing the Voting Scheme with weighted voting is the model with the best behavior, as the accuracy shows. For our voting scheme we have applied 3 classifiers according to the Naive Bayes, KNN and Decision Trees. We have obtained the accuracy of them, to introduce the pertinent weight, so the models are ordered in that way to have more relevance in the vote: KNN, Naive Bayes and Decision Trees. So deep inside we can conclude that KNN has been the one with more relevance in the decision for the votes than the two remaining as the weight dictates.