PlayeRank weighter module

An alternative to the Support Vector Machine algorithm

Project for the course of Machine learning in the Master degree of

Data science for Economics

UNIVERSITÀ DEGLI STUDI DI MILANO

Leonardo Acquaroli

August 30, 2023

Contents

Abstract

This project report presents an alternative approach aimed at enhancing the feature weighting module of PlayeRank, a cutting-edge algorithm designed for ranking football players. PlayeRank's feature weighting module employs a Linear Support Vector Classifier to assign importance scores to various player attributes. This project focuses on the development of alternative learning algorithms that can potentially solve more accurately the underlying classification problem and yield more reliable feature weights.

Introduction

Is it better Leo Messi or Cristiano Ronaldo? Did Olivier Giroud play better than Lautaro Martínez in that game? And in the entire season? Football fans from each part of the world ask these questions every week and PlayeRank tries to give them an objective answer.

PlayeRank [4] is a project developed by P. Cintia and L. Pappalardo that aims to create a ranking, role by role, of football players. The procedure involves four components: a) collection of the soccer events data, b) rating, c) learning, d) ranking.

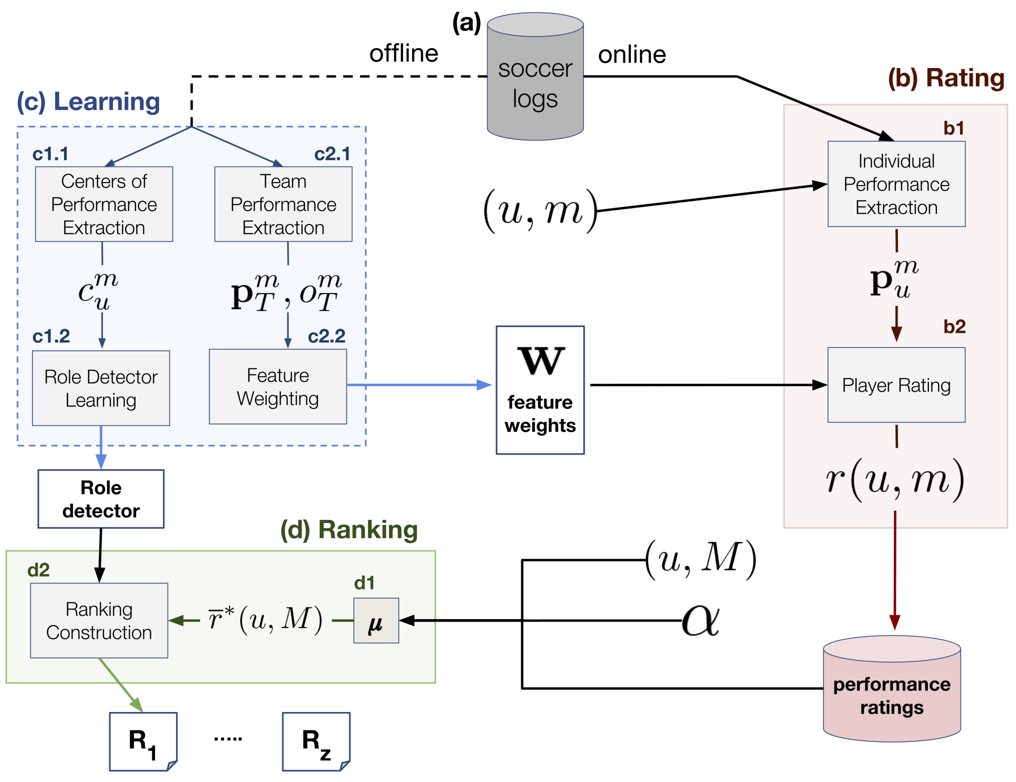


Figure 1: Schema of the PlayeRank framework. Starting from a database of soccer-logs (or events data) (a), it consists of three main phases. The learning phase (c) is an "online" procedure: It must be executed at least once before the other phases, since it generates information used in the other two phases, but then it can be updated separately. The rating (b) and the ranking phases (d) are online procedures, i.e., they are executed every time anew match is available in the database of soccer-logs.

In this project report the focus is on the Team performance extraction and Feature weighting steps (step *c2.1* and *c2.2* of Fig. 1). Indeed, an approach that employs Logistic regression is proposed as an alternative to the Linear Support Vector Classifier used by the authors. The algorithm is then evaluated (with respect both to the paper-reported benchmark metrics and to the custom implementation of two nonparametric algorithms, a kNN and a Decision tree) in order to see if it can beat the performance of the established one.
The report is divided into four sections:

1. In the first section data are presented and the classification problem to solve is outlined

2. In the second section the current (SVM) and proposed (Logistic regression) approaches together with the nonparametric algorithms serving as benchmark are explained theoretically with a look to the Python implementation

3. The third is the models evaluation section

4. In the fourth the weights computed with the alternative algorithm are shown

5. In the fifth section conclusions are drawn.

1 Data and classification problem

1.1 Data

The dataset utilized was published by the same Cintia and Pappalardo in collaboration with the sport-tech company Wyscout [3] and is a collection, for the five main leagues (Serie A, Bundesliga, La Liga, Premier League, Ligue 1) plus the 2018 World cup and the 2016 European championship, of all the events occurring in the matches of the 2017/2018 season. To make it clearer, an event in football data is any action performed by a player that can be recorded and annotated in a row of a table with a lot of context information. For instance a pass, a tackle, a shot or a save and their positions in the pitch, the outcome and possibly the player who received the ball (for passes).

It needs to be pointed out that the data on which the scientists made their calculation on are of the same type and provided by the same company but comprehend an extended (more seasons) version of those used in this project.

The raw event data have been aggregated and processed using the custom functions wrote by the authors and available, together with the code to execute PlayeRank, at https://github.com/mesosbrodleto/playerank.

The aggregated dataset results in a 3882 $\times$ 63 dataframe of 3882 team data for 1941 matches with 62 features and the binary target column 'victory-defeat' that is 1 when the team won the match and -1 when the team lost or tied. When we talk about features in a match we mean 62 kind of events and the values in the rows are their relative counts in the match, where relative count means the difference between the counts of the two teams. For example in match 1, described in the first two rows of Table 1, Team 1 has made 337 accurate simple passes more than Team 2 and reported a victory.

Pass-Simple pass-accurate	Pass-High pass-accurate	...	Foul-yellow card	Pass-Simple pass-not accurate	victory-defeat
337	9	...	1	34	1
-337	-9	...	-1	-34	-1
521	11	...	5	2	1
-521	-11	...	-5	-2	-1

Table 1: An example of the dataframe rows.

1.2 Classification problem

The preprocessed data are the input for the classifiers that solve the problem of predicting whether a match has been won (1) or not (-1). The solution of such a classification problem should be a vector of weights indicating the negative or positive importance of each feature in order to win a game. For this reason, the considered algorithm has been chosen among the class of linear classifiers, which are able to produce such a features weights vector.

2 The algorithms

The methods used to find the best way to weight the players actions are presented delving both in the theoretical concepts and in the implementation.
In the following two subsections two parametric and two algorithms for features weighting and two nonparametric algorithms for benchmarking purposes are presented. The SVM approach used by Cintia and Pappalardo, a Logistic model as an alternative and two nonparametric algorithms, kNN and Decision trees.

2.1 For features weighting

2.1.1 SVM

A Support Vector Machine (SVM for short) is an algorithm that produces a linear classifier by separating the data points with the maximum margin d-dimensional hyperplane, where d is the number of the data features and the margin is defined as the distance between the separating hyperplane and the points, in both the part of the divided space, that are closest to the decision boundary (the hyperplane). Those points are called Support Vectors and are such that if we removed all the other examples except for the Support Vectors, the SVM solution would not change.
The implementation for a generic training set (i.e. also a non-linearly separable one) of a SVM solves the unconstrained problem:

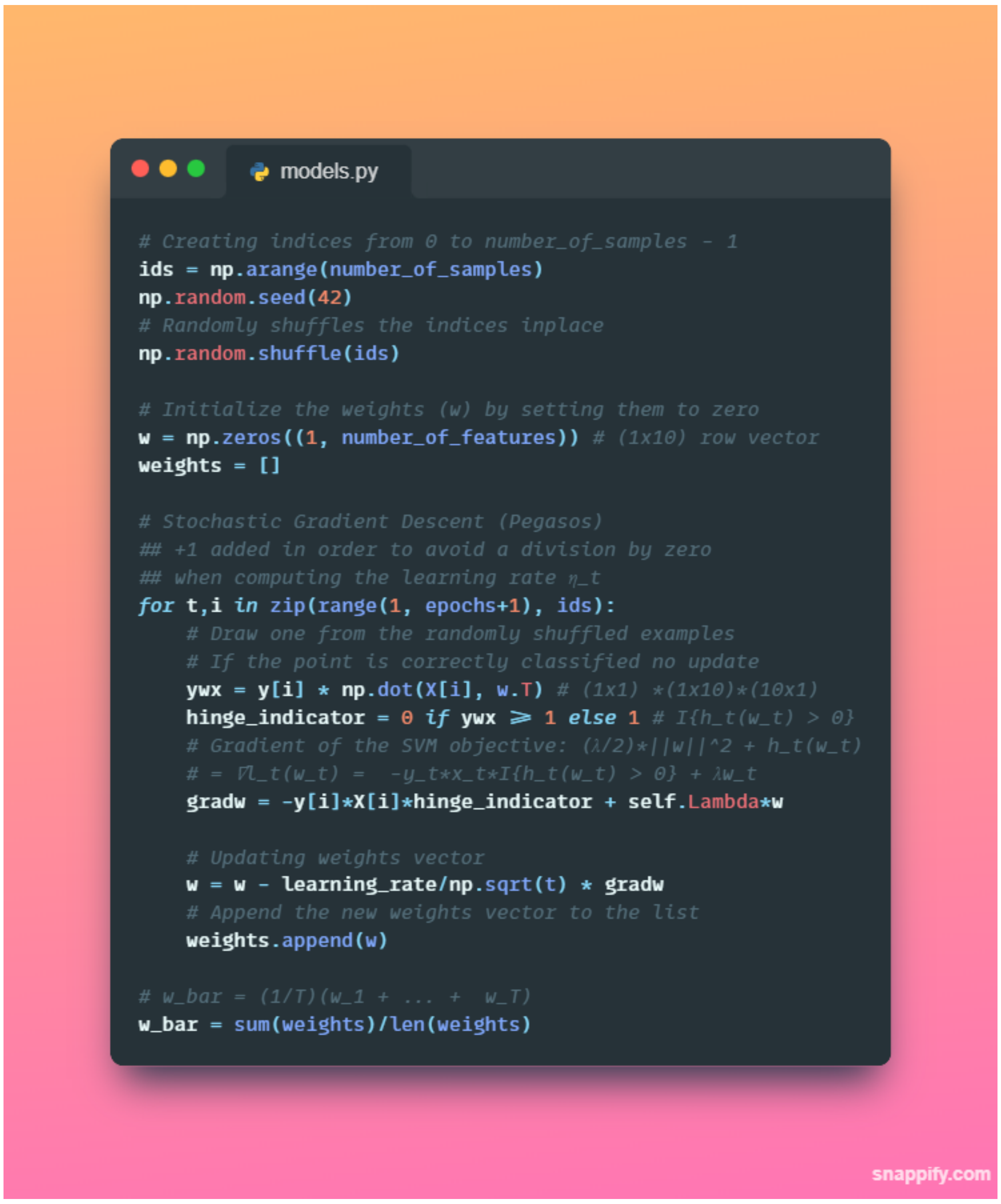
$$\min_{w \in \mathbb{R}^d} F(w)$$

for

$$F(w) = \frac{1}{m} \sum_{t=1}^{m} h_t(w) + \frac{\lambda}{2} \|w\|^2$$

where $h_t(w) = max\{0, 1 - y_t w^T x_t\}$ is the hinge loss, m is the training set size, w is the weights vector and λ is a regularization parameter that balances the two

terms (a sort of conversion rate between the hinge loss and the square norm of w). Unfortunately the minimization task needs a numerical computation technique and in particular an instance of Stochastic Gradient Descent called *Pegasos* has been implemented (Fig. 2).
A Support Vector Machine is what has been used by Cintia and Pappalardo in [4] and is replicated in this project using a custom implementation of the algorithm to analyze its performance on a smaller set of data. The regularization term $\lambda > 0$ has been chosen as the positive value maximizing the AUC and resulted to be $\lambda = 0.063$. Yet, we must notice that the custom implementation is poorer than the industry standards.



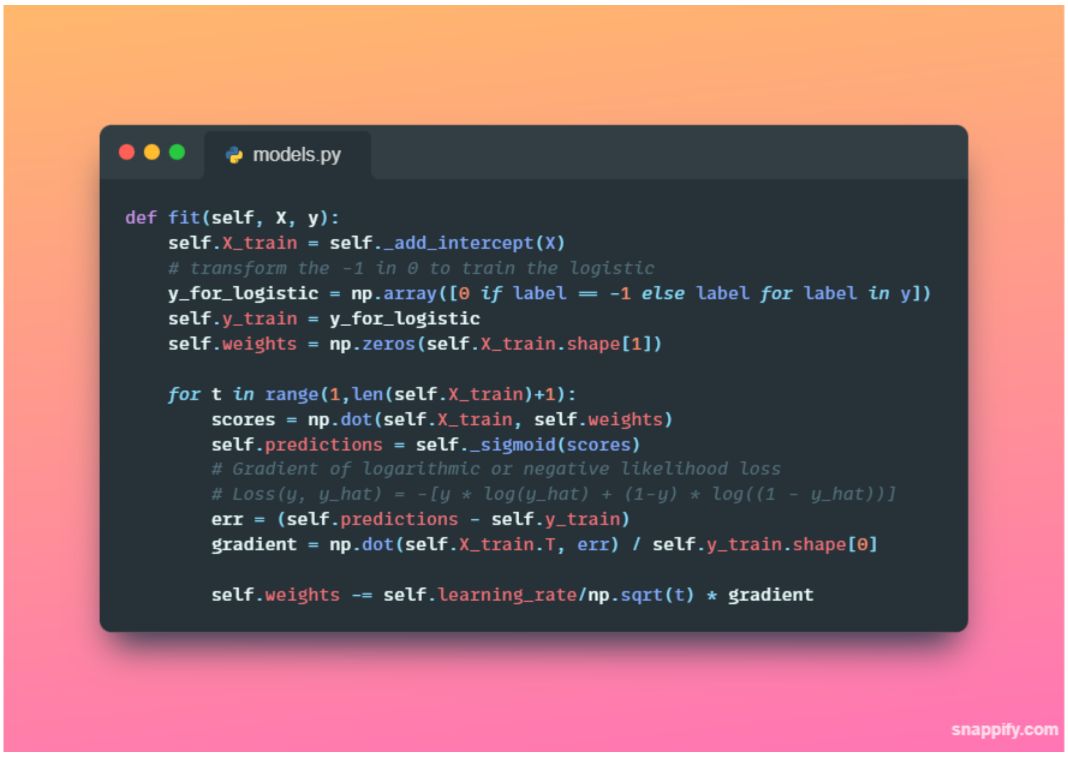
```python
# Creating indices from 0 to number_of_samples - 1
ids = np.arange(number_of_samples)
np.random.seed(42)
# Randomly shuffles the indices inplace
np.random.shuffle(ids)

# Initialize the weights (w) by setting them to zero
w = np.zeros((1, number_of_features)) # (1x10) row vector
weights = []

# Stochastic Gradient Descent (Pegasos)
## +1 added in order to avoid a division by zero
## when computing the learning rate η_t
for t,i in zip(range(1, epochs+1), ids):
    # Draw one from the randomly shuffled examples
    # If the point is correctly classified no update
    ywx = y[i] * np.dot(X[i], w.T) # (1x1) *(1x10)*(10x1)
    hinge_indicator = 0 if ywx ⩾ 1 else 1 # I{h_t(w_t) > 0}
    # Gradient of the SVM objective: (λ/2)*||w||^2 + h_t(w_t)
    # = ∇l_t(w_t) =  -y_t*x_t*I{h_t(w_t) > 0} + λw_t
    gradw = -y[i]*X[i]*hinge_indicator + self.Lambda*w

    # Updating weights vector
    w = w - learning_rate/np.sqrt(t) * gradw
    # Append the new weights vector to the list
    weights.append(w)

# w_bar = (1/T)(w_1 + ... +  w_T)
w_bar = sum(weights)/len(weights)
```

Figure 2: Pegasos implementation in the SVM class of the models.py file.

2.1.2 Logistic regression

Logistic regression is a technique that instead of producing a vector of binary predictions outputs a vector of probabilities that are then converted to binary guesses using a threshold of 0.5. Logistic regression is a generalized linear model and uses the sigmoid function $\sigma(w^\top x) = \frac{1}{1+e^{-w^\top x}} \in (0,1)$ to transform the linear predictions $w^\top x$ into a range $(0,1)$ of values.

The implementation of the algorithm minimizes the negative log-likelihood loss function $l(y, \hat{y}) = -\left[y \log(\hat{y}) + (1-y)\log(1-\hat{y})\right]$ through gradient descent using a learning rate $\eta_t = \frac{0.01}{\sqrt{t}}$ where $t = 1, ..., T$ is the number of each update.



```python
def fit(self, X, y):
    self.X_train = self._add_intercept(X)
    # transform the -1 in 0 to train the logistic
    y_for_logistic = np.array([0 if label == -1 else label for label in y])
    self.y_train = y_for_logistic
    self.weights = np.zeros(self.X_train.shape[1])

    for t in range(1,len(self.X_train)+1):
        scores = np.dot(self.X_train, self.weights)
        self.predictions = self._sigmoid(scores)
        # Gradient of logarithmic or negative likelihood loss
        # Loss(y, y_hat) = -[y * log(y_hat) + (1-y) * log((1 - y_hat))]
        err = (self.predictions - self.y_train)
        gradient = np.dot(self.X_train.T, err) / self.y_train.shape[0]

        self.weights -= self.learning_rate/np.sqrt(t) * gradient
```

Figure 3: The fit() method in which the gradient descent updates are implemented for the logistic model.

2.2 For benchmarking

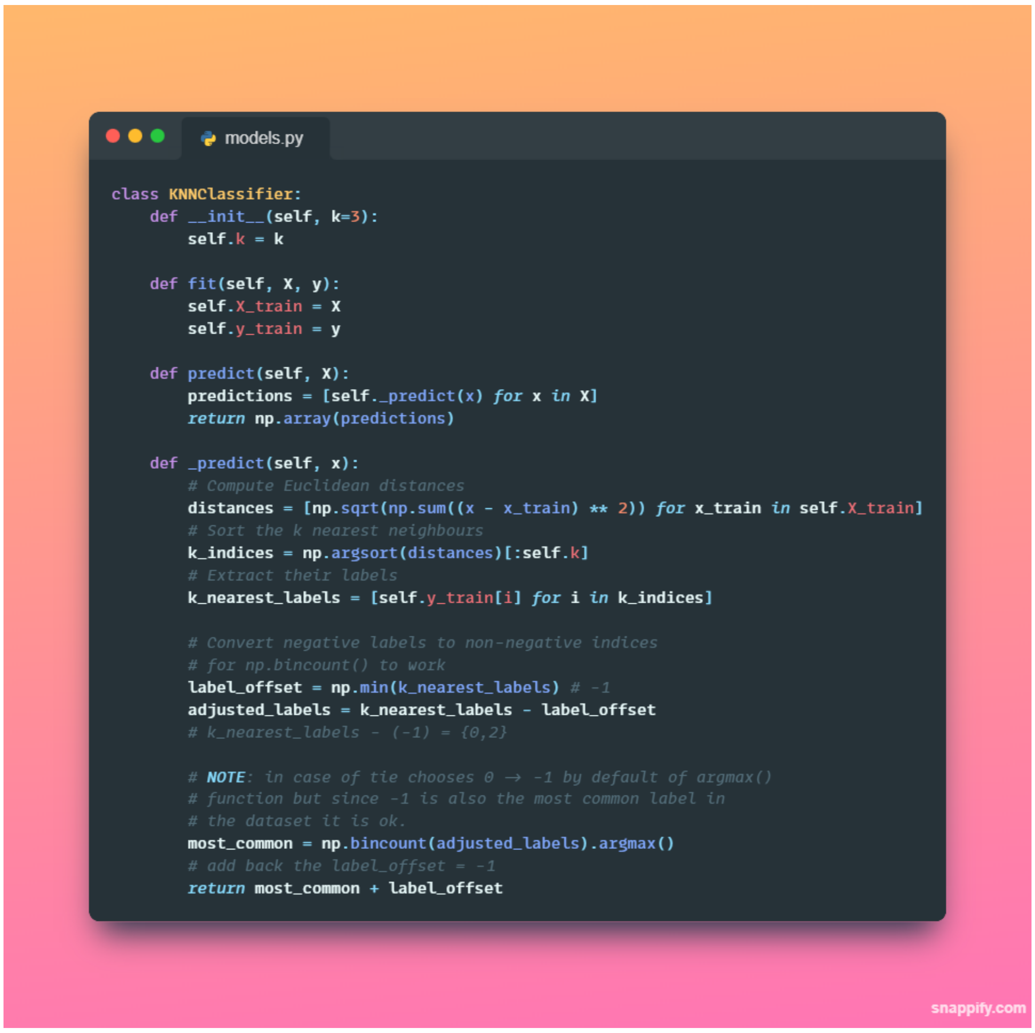
2.2.1 kNN

The k-Nearest Neighbors (kNN) algorithm is a nonparametric method here used to classify the matches outcome. kNN operates under the premise that matches with proximate players feature distributions are likely to share analogous outcome and thus classifies a new point by following those steps:

1. Calculates the Euclidean distance with the unlabelled and the training points and finds the k nearest observations (neighbours)

2. Assigns the label to the new point corresponding to the label of the majority of the k nearest neighbours.

In case of k even and a tie in the labels, the custom implementation chooses $\hat{y} = -1$ that is the most frequent label in the dataset (due to draws). In the current analysis the best k has been chosen as the one which maximizes the AUC and resulted to be $k = 7$.



```python
class KNNClassifier:
    def __init__(self, k=3):
        self.k = k

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def predict(self, X):
        predictions = [self._predict(x) for x in X]
        return np.array(predictions)

    def _predict(self, x):
        # Compute Euclidean distances
        distances = [np.sqrt(np.sum((x - x_train) ** 2)) for x_train in self.X_train]
        # Sort the k nearest neighbours
        k_indices = np.argsort(distances)[:self.k]
        # Extract their labels
        k_nearest_labels = [self.y_train[i] for i in k_indices]

        # Convert negative labels to non-negative indices
        # for np.bincount() to work
        label_offset = np.min(k_nearest_labels) # -1
        adjusted_labels = k_nearest_labels - label_offset
        # k_nearest_labels - (-1) = {0,2}

        # NOTE: in case of tie chooses 0 → -1 by default of argmax()
        # function but since -1 is also the most common label in
        # the dataset it is ok.
        most_common = np.bincount(adjusted_labels).argmax()
        # add back the label_offset = -1
        return most_common + label_offset
```

Figure 4: The custom implementation of kNN.
The default k value is set to $k = 3$.

2.2.2 Decision trees

Decision trees are a class of nonparametric learning algorithms that learn a predictor by performing different splits of the training set each based on a feature and a threshold until there is no more information gain after a split or it reaches the maximum depth (i.e. number of levels the tree can have from the root to the farthest leaf node) which is a user-defined hyperparameter controlling the complexity of the tree and preventing overfitting. The criteria usually adopted to measure the information gain are three: the Gini entropy, the Scaled entropy and a square root form of entropy. In this project's implementation the Scaled entropy $\psi = -\frac{p}{2}\log_2(p) - \frac{1-p}{2}\log_2(1-p)$ has been used. For what concerns the maximum depth hyperparameter, the fine-tuning process performed by maximizing the AUC resulted in a maximum depth of 4 levels. This kind of algorithms are called trees because their decision path can be drawn down and easily interpreted even if the output is not a series of parameters. What follows is a representation of the Decision tree classifier developed in this project.

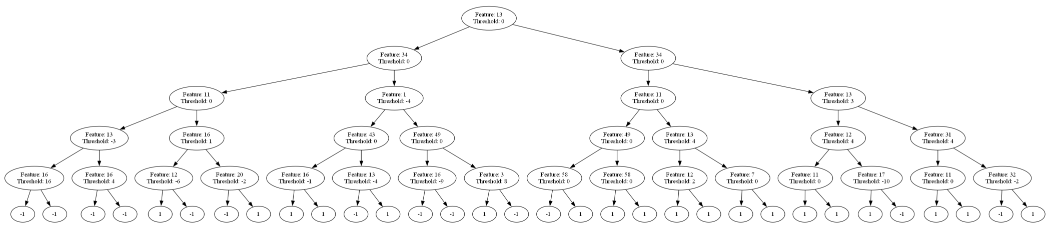
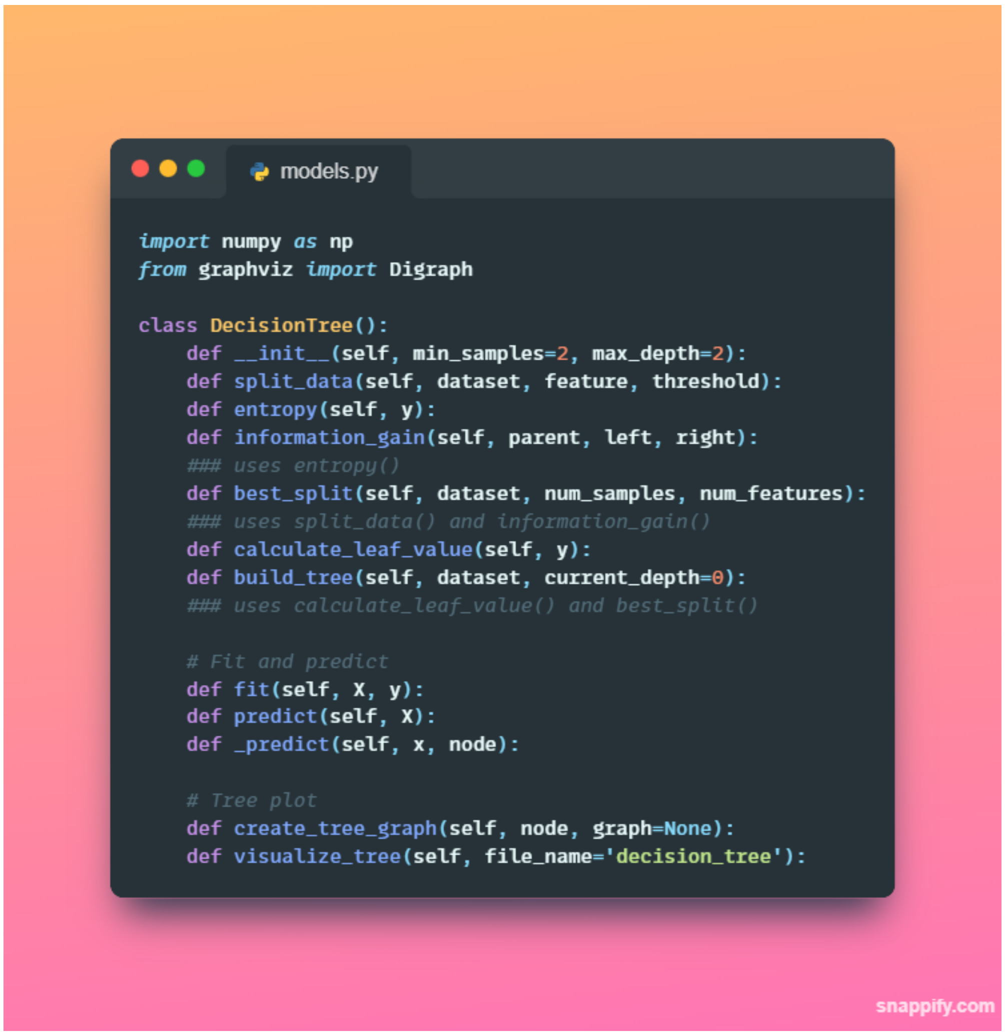


Figure 5: Decision tree classifier diagram.

Unlike the implementations of the other algorithms, the tree's one needed several functions each of them performing a specific action. The details of the code structure are reported in Fig. 6.



```python
import numpy as np
from graphviz import Digraph

class DecisionTree():
    def __init__(self, min_samples=2, max_depth=2):
    def split_data(self, dataset, feature, threshold):
    def entropy(self, y):
    def information_gain(self, parent, left, right):
    ### uses entropy()
    def best_split(self, dataset, num_samples, num_features):
    ### uses split_data() and information_gain()
    def calculate_leaf_value(self, y):
    def build_tree(self, dataset, current_depth=0):
    ### uses calculate_leaf_value() and best_split()

    # Fit and predict
    def fit(self, X, y):
    def predict(self, X):
    def _predict(self, x, node):

    # Tree plot
    def create_tree_graph(self, node, graph=None):
    def visualize_tree(self, file_name='decision_tree'):
```

Figure 6: Structure of the Decision tree implementation.

3 Models evaluation

In this section the performance of the Support Vector Classifier and the Logistic regression are compared to two benchmarks: the metrics reported by the authors and the performance of the two nonparametric algorithms: the kNN and the Decision tree. Consistently with the metrics reported by Cintia and Pappalardo the comparison will focus on three performance metrics:

- AUC: the Area Under the Receiver Operating Characteristic Curve (AUC) that is a way to measure the price paid for the precision-recall trade-off [1]
- F1: That is the harmonic average of precision and recall [5]
- Accuracy: The fraction of correctly predicted examples over the number of all the observations. [2]

The above-mentioned metrics are obtained as the mean values of a 5-fold cross validation run over the entire dataset for each algorithm.

3.1 Authors reported metrics

The authors reported $AUC = 0.89$, $F1 = 0.81$, $Accuracy = 0.82$ after using a 5-fold cross validation on 15'695 matches examples.

3.2 Nonparametric benchmarks

Both the nonparametric benchmarks have been cross validated on the entire dataset of 3882 matches and these are their results:

kNN The metrics for the 7NN custom implementation are: $AUC = 0.63$, $F1 = 0.51$, $Accuracy = 0.67$.

Decision tree The metrics for the Decision tree are: $AUC = 0.72$, $F1 = 0.64$, $Accuracy = 0.74$.

3.3 SVM and Logistic regression metrics

SVM Though it was the algorithm originally chosen by the authors of PlayeRank, its custom implementation together with the smaller size of the training set badly influenced its performance which is summarised by the following figures: $AUC = 0.69$, $F1 = 0.64$, $Accuracy = 0.68$

Logistic regression The Logistic model, instead, is the one with the best performance among the custom implemented algorithms of this project. Indeed, it overcomes the nonparametric benchmarks and the SVM metrics with $AUC = 0.77$, $F1 = 0.72$, $Accuracy = 0.77$. Yet, it is still far from the metrics of the weighting module of PlayeRank.

4 Features weights

After having found an alternative algorithm to the custom implemented Linear Support Vector Classifier, the last step is to extract and show the features weights obtained from the Logistic regression. These are shown in Fig. 7 from which we can see that the most positively weighted feature is the ground loose ball duel won which is a duel in which the ball is neither in the possession of the attacker nor in that of the defender and the ball is won by the team to which the row refers to.

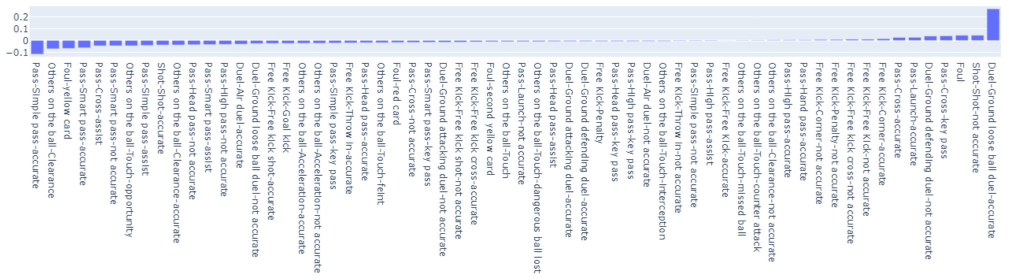


Figure 7: Players actions importance as weighted by the Logistic regression.

5 Conclusions

In this project, the focus was on enhancing the feature weighting module of PlayeRank, a football players ranking algorithm, through alternative learning algorithms. The objective was to identify an algorithm that could provide more accurate feature weights and potentially improve the ranking accuracy.
The performance evaluation revealed that the Support Vector Machine (SVM) algorithm, while originally employed in PlayeRank, demonstrated suboptimal performances in the custom implementation. On the other hand, Logistic Regression showed promising results, outperforming both the SVM and the non-parametric benchmarks, k Nearest Neighbors (kNN) and Decision Trees. However, the Logistic Regression model's performance was not yet on par with the metrics reported by the authors for the PlayeRank feature weighting module.
Additionally, the analysis of feature importance weights generated by the Logistic Regression model highlighted the significance of specific player actions in determining match outcomes. Notably, the ground loose ball duel won emerged as the most positively weighted feature, suggesting its strong influence on the team's success.

References

[1] Andrew P. Bradley. "The use of the area under the ROC curve in the evaluation of machine learning algorithms". In: *Pattern Recognition* 30.7 (1997), pp. 1145–1159. ISSN: 0031-3203. DOI: https://doi.org/10.1016/S0031-3203(96)00142-2. URL: https://www.sciencedirect.com/science/article/pii/S0031320396001422.

[2] Brian Liu and Madeleine Udell. "Impact of Accuracy on Model Interpretations". In: *CoRR* abs/2011.09903 (2020). arXiv: 2011.09903. URL: https://arxiv.org/abs/2011.09903.

[3] Luca Pappalardo et al. "A public data set of spatio-temporal match events in soccer competitions". In: *Scientific Data* 6 (Oct. 2019). DOI: 10.1038/s41597-019-0247-7.

[4] Luca Pappalardo et al. "PlayeRank". In: *ACM Transactions on Intelligent Systems and Technology* 10.5 (2019), pp. 1–27. DOI: 10.1145/3343172. URL: https://doi.org/10.1145\%2F3343172.

[5] David M. W. Powers. "Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation". In: *CoRR* abs/2010.16061 (2020). arXiv: 2010.16061. URL: https://arxiv.org/abs/2010.16061.