

✓ 1. Problem Statement

The Dataset

This Dataset is called 'Comprehensive FIFA Football Players Dataset', with the csv file named 'fifa_players', and has been obtained from kaggle :

(<https://www.kaggle.com/datasets/maso0dahmed/football-players-data/data>)

This Dataset contains over 17,000 entries of football/soccer players, attributes, performance metrics, and market values, with all the attributes provided by Fifa themselves, which is the international governing body of association football. These include player names, full names, dates of birth, ages, heights, weights, playing positions, nationalities, FIFA overall ratings, potential ratings, market values, weekly wages, preferred foot, international reputation, weak foot ability, skill moves, body types, release clauses, and detailed performance metrics such as crossing, finishing, dribbling, and more.

Building a machine learning with this dataset can significantly enhance determining the real and true transfer market values of players, ensuring that teams don't overpay for players, and also for teams to not sell a player for a price lower than what they are worth. It also would allow teams to identify players that are undervalued in the market, providing an advantage in acquiring a talent at lower costs. By analyzing some of the variables listed above, a machine learning model can identify patterns and correlations that human analysis might overlook, providing a more accurate, objective, and data-driven valuation of players. The model can adapt over time, incorporating new data as player performances evolve across seasons. This means that the model's predictions remain relevant and accurate, reflecting the dynamic nature of football's transfer market. In terms of young players, who aren't as developed, the model also takes into account each players' potential to combat this possible issue

First lets import all the libraries that I will use in this Notebook

```
# Importing All Required Libraries

import pandas as pd
import numpy as np

import seaborn as sns
from matplotlib import pyplot as plt

from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
from sklearn.linear_model import LinearRegression, LassoCV, RidgeCV
from sklearn.tree import DecisionTreeRegressor, plot_tree
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# Suppression of warning messages
import warnings
warnings.filterwarnings('ignore', category=UserWarning)

%matplotlib inline
```


✓ 2. Exploring the Data

First we load the csv file 'fifa_players.csv' into the notebook, into a pandas dataframe. Then I will display the first 5 rows to show the structure of the dataset, and see how many rows and columns there are

```
# Read the CSV file 'fifa_players.csv', load the data set into a DataFrame
df = pd.read_csv('fifa_players.csv')

# Print the shape of the DataFrame
print("Shape of the data= ", df.shape)

# Display the first five rows
df.head()
```

 Shape of the data= (17954, 51)

	name	full_name	birth_date	age	height_cm	weight_kgs	positions	nat
0	L. Messi	Lionel Andrés Messi Cuccittini	6/24/1987	31	170.18	72.1	CF,RW,ST	
1	C. Eriksen	Christian Dannemann Eriksen	2/14/1992	27	154.94	76.2	CAM,RM,CM	
2	P. Pogba	Paul Pogba	3/15/1993	25	190.50	83.9	CM,CAM	
3	L. Insigne	Lorenzo Insigne	6/4/1991	27	162.56	59.0	LW,ST	
4	K. Koulibaly	Kalidou Koulibaly	6/20/1991	27	187.96	88.9	CB	

5 rows x 51 columns

```
df.columns # Listing the names of all the columns in the dataset
```


```
⇒ Index(['name', 'full_name', 'birth_date', 'age', 'height_cm',  
        'weight_kgs',  
        'positions', 'nationality', 'overall_rating', 'potential',  
        'value_euro',  
        'wage_euro', 'preferred_foot', 'international_reputation(1-5)',  
        'weak_foot(1-5)', 'skill_moves(1-5)', 'body_type',  
        'release_clause_euro', 'national_team', 'national_rating',  
        'national_team_position', 'national_jersey_number', 'crossing',  
        'finishing', 'heading_accuracy', 'short_passing', 'volleys',  
        'dribbling', 'curve', 'freekick_accuracy', 'long_passing',  
        'ball_control', 'acceleration', 'sprint_speed', 'agility',  
        'reactions',  
        'balance', 'shot_power', 'jumping', 'stamina', 'strength',  
        'long_shots',  
        'aggression', 'interceptions', 'positioning', 'vision',  
        'penalties',  
        'composure', 'marking', 'standing_tackle', 'sliding_tackle'],  
        dtype='object')
```

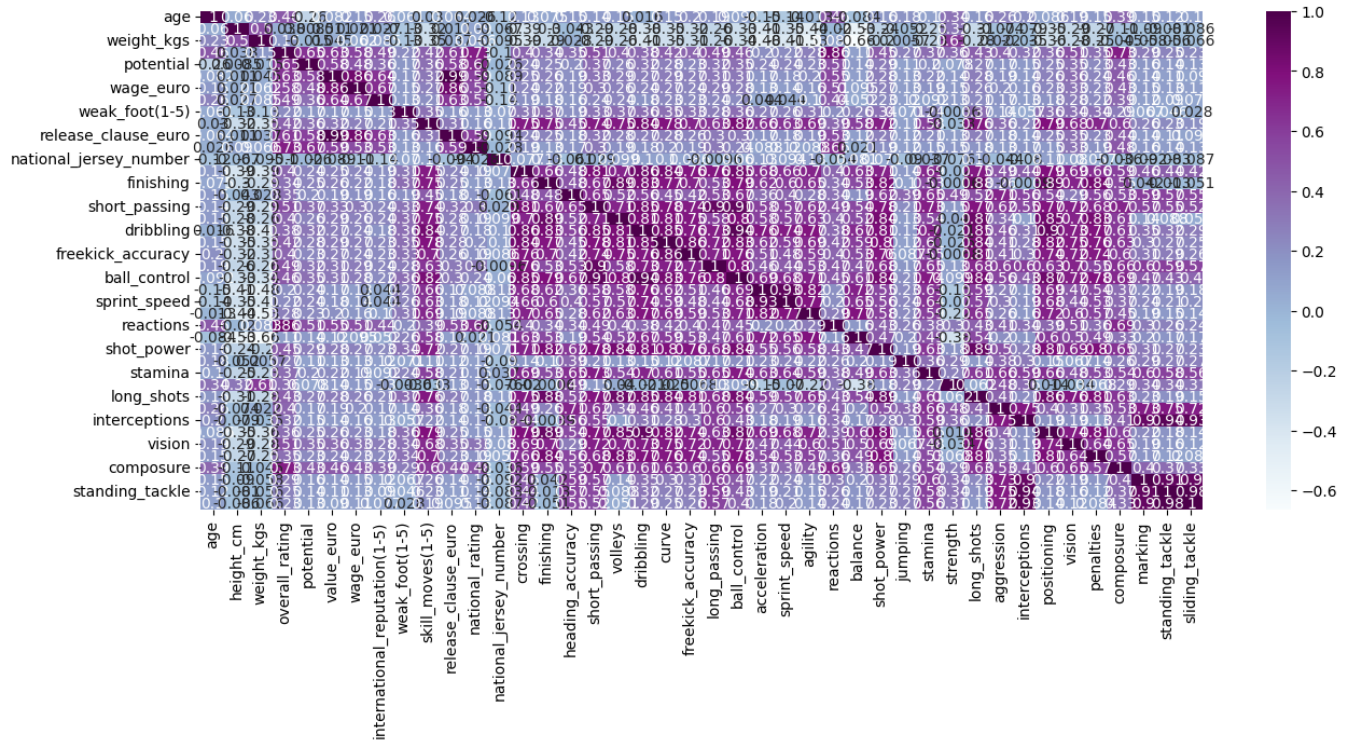
```
df.dtypes
```

```
⇒ name object
full_name object
birth_date object
age int64
height_cm float64
weight_kgs float64
positions object
nationality object
overall_rating int64
potential int64
value_euro float64
wage_euro float64
preferred_foot object
international_reputation(1-5) int64
weak_foot(1-5) int64
skill_moves(1-5) int64
body_type object
release_clause_euro float64
national_team object
national_rating float64
national_team_position object
national_jersey_number float64
crossing int64
finishing int64
heading_accuracy int64
short_passing int64
volleys int64
dribbling int64
curve int64
freekick_accuracy int64
long_passing int64
ball_control int64
acceleration int64
sprint_speed int64
agility int64
reactions int64
balance int64
shot_power int64
jumping int64
stamina int64
strength int64
long_shots int64
aggression int64
interceptions int64
positioning int64
vision int64
penalties int64
composure int64
marking int64
standing_tackle int64
sliding_tackle int64
dtype: object
```

Here is the correlation matrix between all the input features in the dataset. As you can see, because there are many features, the correlation matrix is harder to read and analyse. Therefore, I have provided another correlation matrix where only the target variable ('value_euro') is on the x axis, and all the other variables in the y axis. This makes it easier to compare the correlations of each feature with the target variable

```
corr=df.corr() # gives us the correlation values
plt.figure(figsize=(15,6))
sns.heatmap(corr, annot = True, cmap="BuPu") # let's visualise the correlation
plt.show()
```

 <ipython-input-58-1d069d1c0964>:1: FutureWarning: The default value of nume
corr=df.corr() # gives us the correlation values




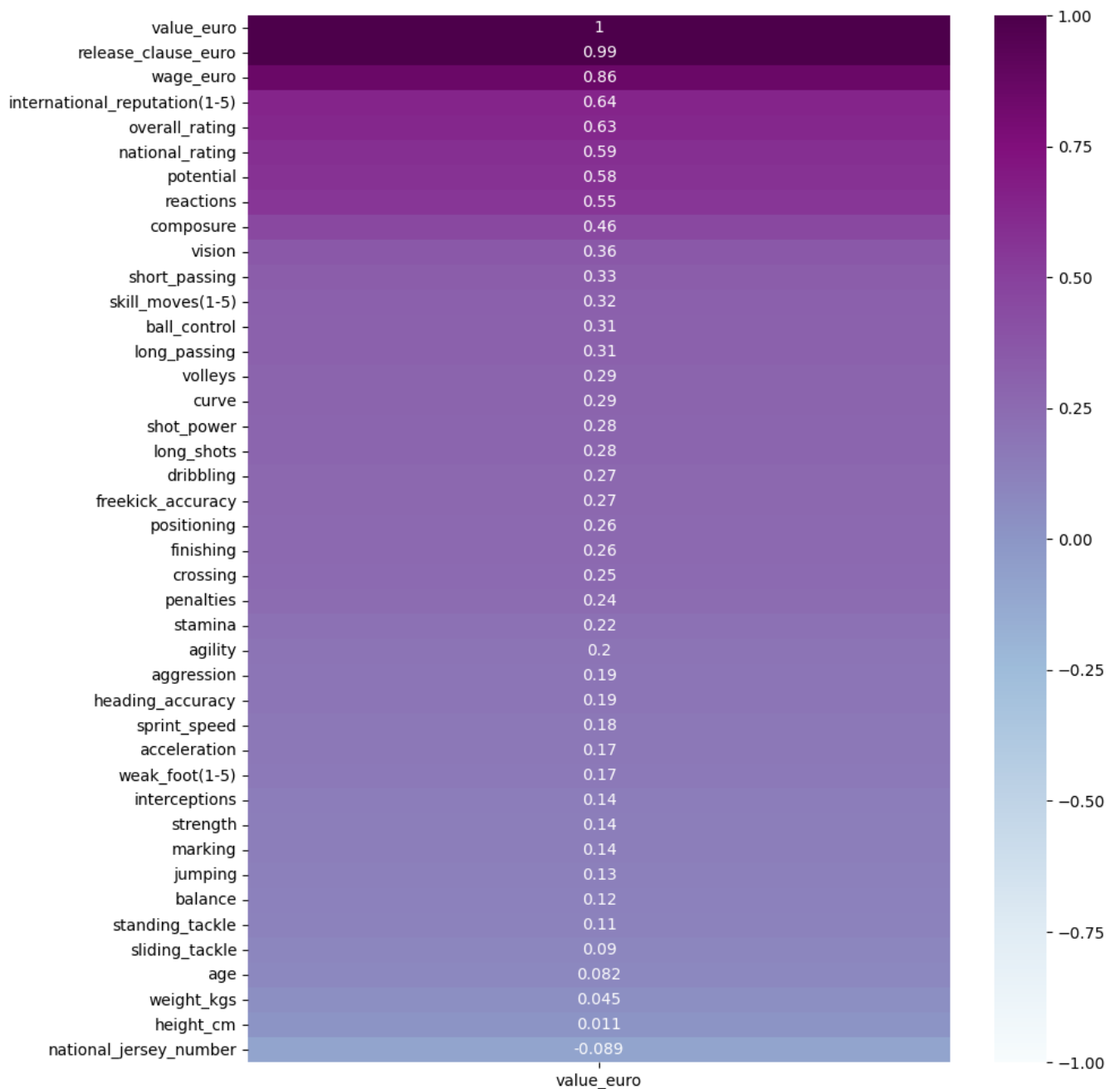
```
corr = df.corr()
```

```
# Isolate the 'value_euro' column
value_euro_corr = corr[['value_euro']]

# Plot the correlations with 'value_euro'
plt.figure(figsize=(10, 12))
sns.heatmap(value_euro_corr.sort_values(by='value_euro', ascending=False),
            annot=True,
            cmap="BuPu",
            cbar=True,
            vmin=-1,
            vmax=1)

plt.show()
```

 <ipython-input-59-fcffb3648838>:1: FutureWarning: The default value of nume.
corr = df.corr()



✓ 3. Pre-processing the data

Feature Engineering: Removing unnecessary columns, as well as combining columns together that are part of the same category e.g. skills, defence, physical, attack

```
df['physical'] = (df["strength"] + df["sprint_speed"] + df["agility"] + df["rea
df["jumping"] + df["acceleration"] + df["aggression"])/8

df['defence_rating'] = (df["sliding_tackle"] + df["standing_tackle"] + df["inte
df["positioning"])/5

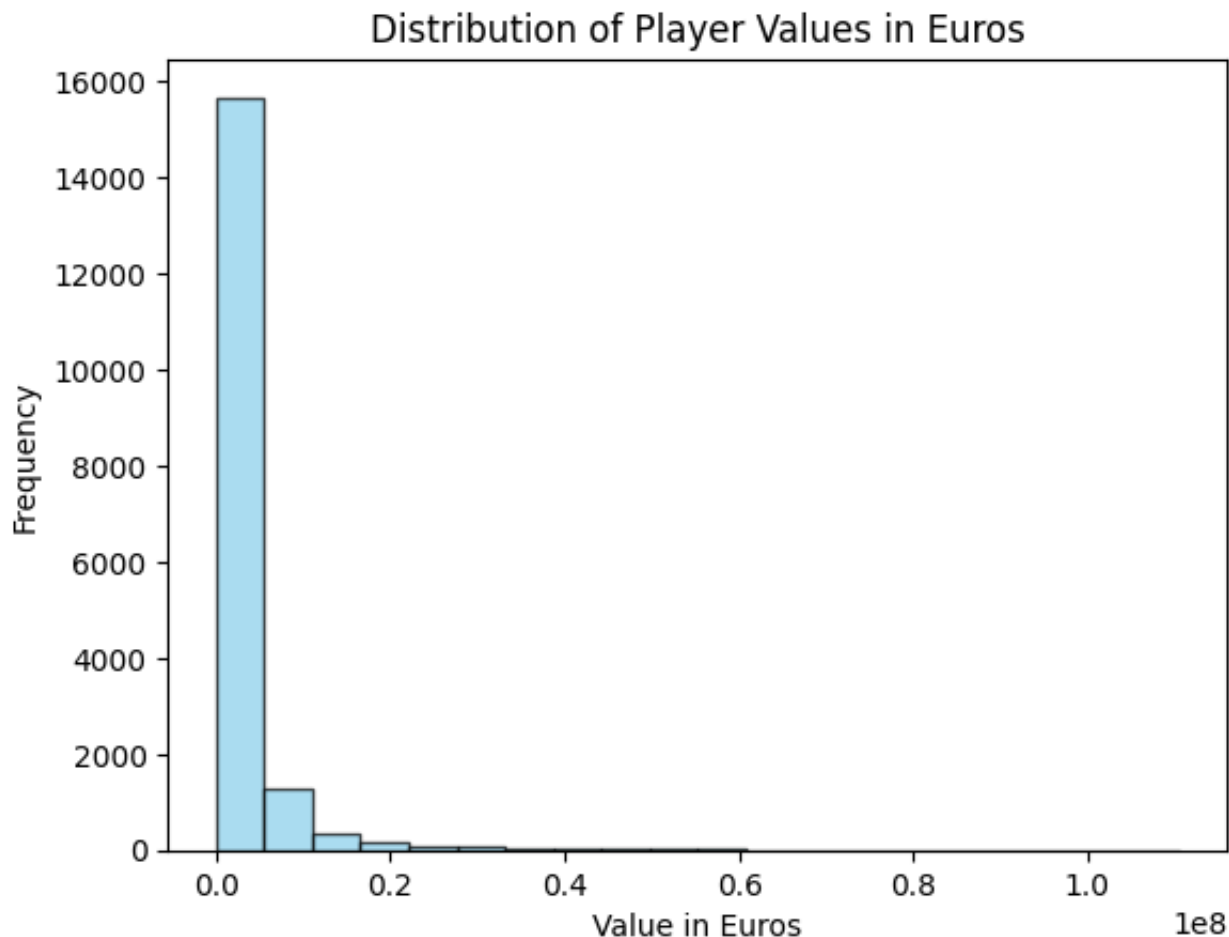
df["skills"] = (df["ball_control"] + df["short_passing"] + df["long_passing"] +
df["vision"] + df["dribbling"] + df["balance"])/7

df["attack_rating"] = (df["crossing"] + df["finishing"] + df["long_shots"] + df
df["heading_accuracy"] + df['curve']+ df['freekick_a
```

```
plt.hist(df["value_euro"], bins=20, alpha=0.7, color='skyblue', edgecolor='black')

# Adding labels and title
plt.xlabel('Value in Euros')
plt.ylabel('Frequency')
plt.title('Distribution of Player Values in Euros')
```

```
plt.show()
```



From this Histogram, we can see that the distribution is heavily right-skewed, meaning that most of the player values are concentrated in the lower end of the euro value spectrum, with very few players having high values. The bins are concentrated towards the left of the plot, indicating that there are very few high-value outliers.

```
# Applying a logarithmic transformation to 'value_euro' column
# Ensure there are no zero or negative values in 'value_euro' before this trans
df["value_euro"] = np.log(df["value_euro"])

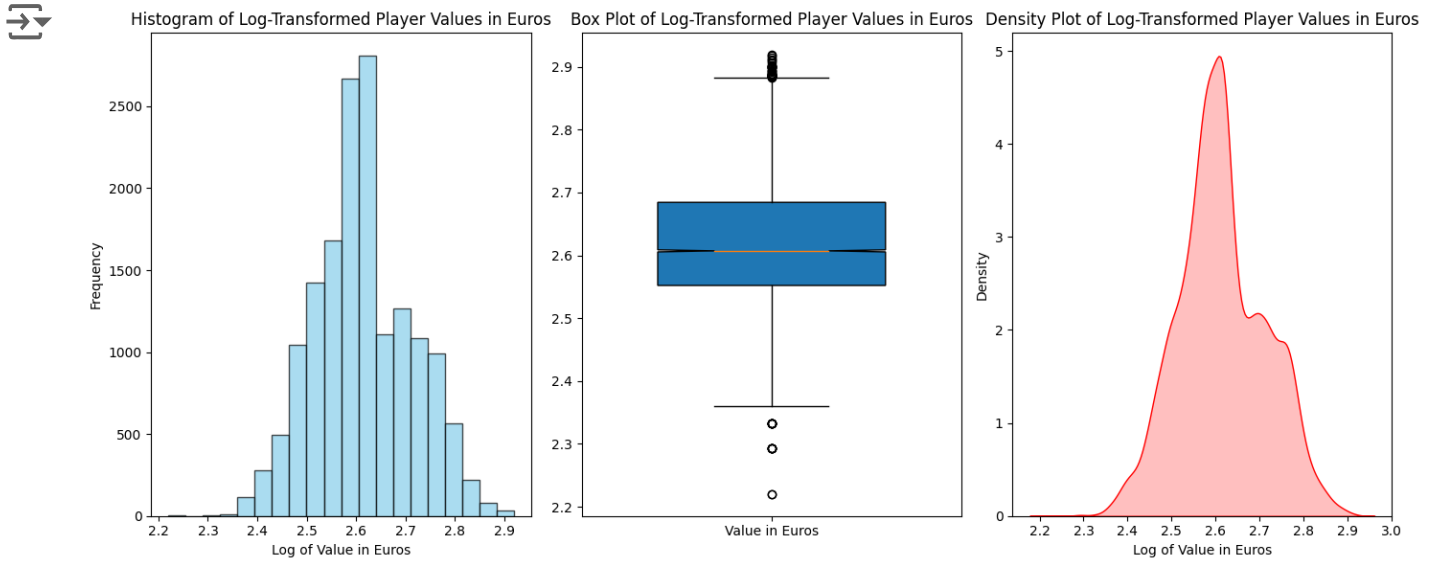
# Creating a histogram for the 'value_euro' column with customized bin colors
plt.figure(figsize=(14, 6))
```

```
# Histogram
plt.subplot(1, 3, 1)
plt.hist(df["value_euro"], bins=20, alpha=0.7, color='skyblue', edgecolor='black')
plt.xlabel('Log of Value in Euros')
plt.ylabel('Frequency')
plt.title('Histogram of Log-Transformed Player Values in Euros')

# Box Plot
plt.subplot(1, 3, 2)
plt.boxplot(df["value_euro"], vert=True, patch_artist=True, notch=True, widths=0.8)
plt.xticks([1], ['Value in Euros'])
plt.title('Box Plot of Log-Transformed Player Values in Euros')

# Density Plot
plt.subplot(1, 3, 3)
sns.kdeplot(df["value_euro"], fill=True, color="r")
plt.xlabel('Log of Value in Euros')
plt.ylabel('Density')
plt.title('Density Plot of Log-Transformed Player Values in Euros')

plt.tight_layout()
plt.show()
```



To address this, a logarithmic transformation is applied to the 'value_euro' column, as shown in the second histogram. The logarithmic transformation is a common technique used to reduce skewness in positively skewed data. By applying a log transformation, we can often convert a skewed distribution to one that is more symmetric and that more closely resembles a normal distribution.

The second histogram shows the distribution of the log-transformed 'value_euro' values. This plot indicates a more symmetric distribution of player values, which is less skewed than the original data. It spreads out the values so that we can better understand the distribution and is particularly useful for handling outliers, as it reduces the impact of the extreme values on the analysis.

The box plot of the log-transformed values shows a distribution that is more compact and less skewed, with fewer outliers. The median is more centered, and the interquartile range (the box) is more symmetric about the median.

The density plot for the log-transformed values shows a smooth curve that approximates the normal distribution, further confirming that the log transformation has helped in normalizing the data.

As we combined some of the columns above, we can now remove the individual columns that aren't necessary for this problem. I have also removed all the goalkeepers, as there's no attributes available in the dataset to assess the quality of the goal keepers, so this model is for all outfield players


```
# Section 5: Data Cleaning – Dropping More Columns & Handling Missing Values
# Combine the two lists of columns to be removed
df = df[~df['positions'].str.contains('GK', na=False)]

columns_remove = [
    'crossing', 'finishing', 'heading_accuracy', 'short_passing', 'volleys',
    'dribbling', 'curve', 'freekick_accuracy', 'long_passing', 'ball_control',
    'acceleration', 'sprint_speed', 'agility', 'reactions', 'balance', 'shot_po
    'jumping', 'stamina', 'strength', 'long_shots', 'aggression', 'interception
    'positioning', 'vision', 'penalties', 'composure', 'marking', 'standing_tac
    'sliding_tackle', 'name', 'full_name', 'birth_date', 'nationality',
    'preferred_foot', 'body_type', "release_clause_euro", "national_team", "nati
    "body_type", "skill_moves(1-5)", "positions", 'height_cm', 'weight_kgs', 'a
]

# Now you can use this combined, deduplicated list to drop columns from your Da
df = df.drop(columns=columns_remove)

# Further filtering to remove goalkeepers from the DataFrame, assuming 'GK' ide

# Display the first few rows of the DataFrame without the specified columns and
df.head()
```

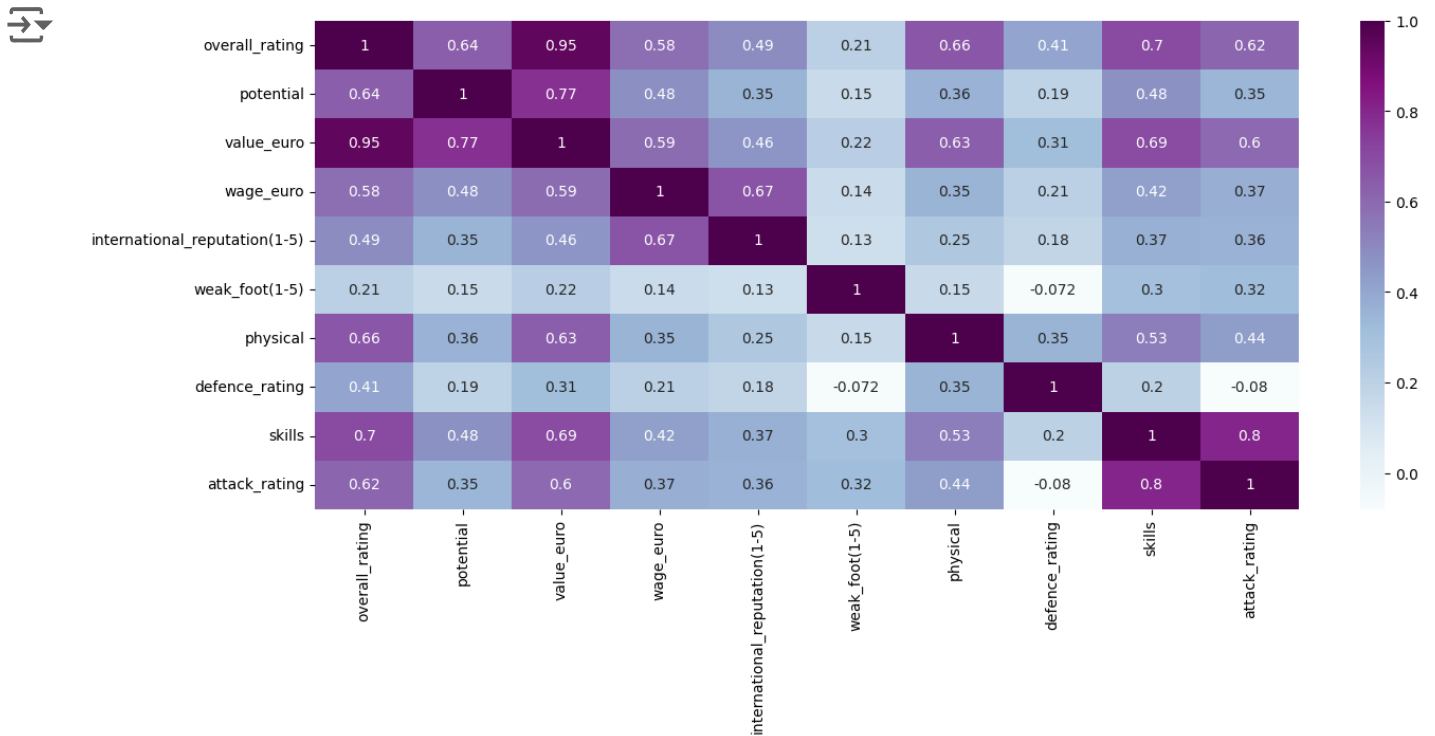


	overall_rating	potential	value_euro	wage_euro	international_reputatio
0	94	94	18.520526	565000.0	
1	88	89	18.056837	205000.0	
2	88	91	18.105970	255000.0	
3	88	88	17.942645	165000.0	
4	88	91	17.909855	135000.0	

Below, is a correlation matrix, that shows how correlated each of the selected features are with the market value of the player. From the heatmap, we can see that overall_rating has the highest positive correlation (0.95) with value_euro, indicating a very strong relationship where higher-rated players tend to have higher market values. This is followed by potential with a correlation of 0.77, suggesting that players with higher potential ratings are also likely to have higher market values. Other features like skills, physical, attack_rating, and wage_euro show varying degrees of positive correlation, signifying that these attributes are positively associated with a player's market value.

The feature defence_rating has a lower positive correlation (0.31), and age has the lowest positive correlation (0.22) among the ones shown, which may indicate these factors have a less direct relationship with the market value of players compared to other features.

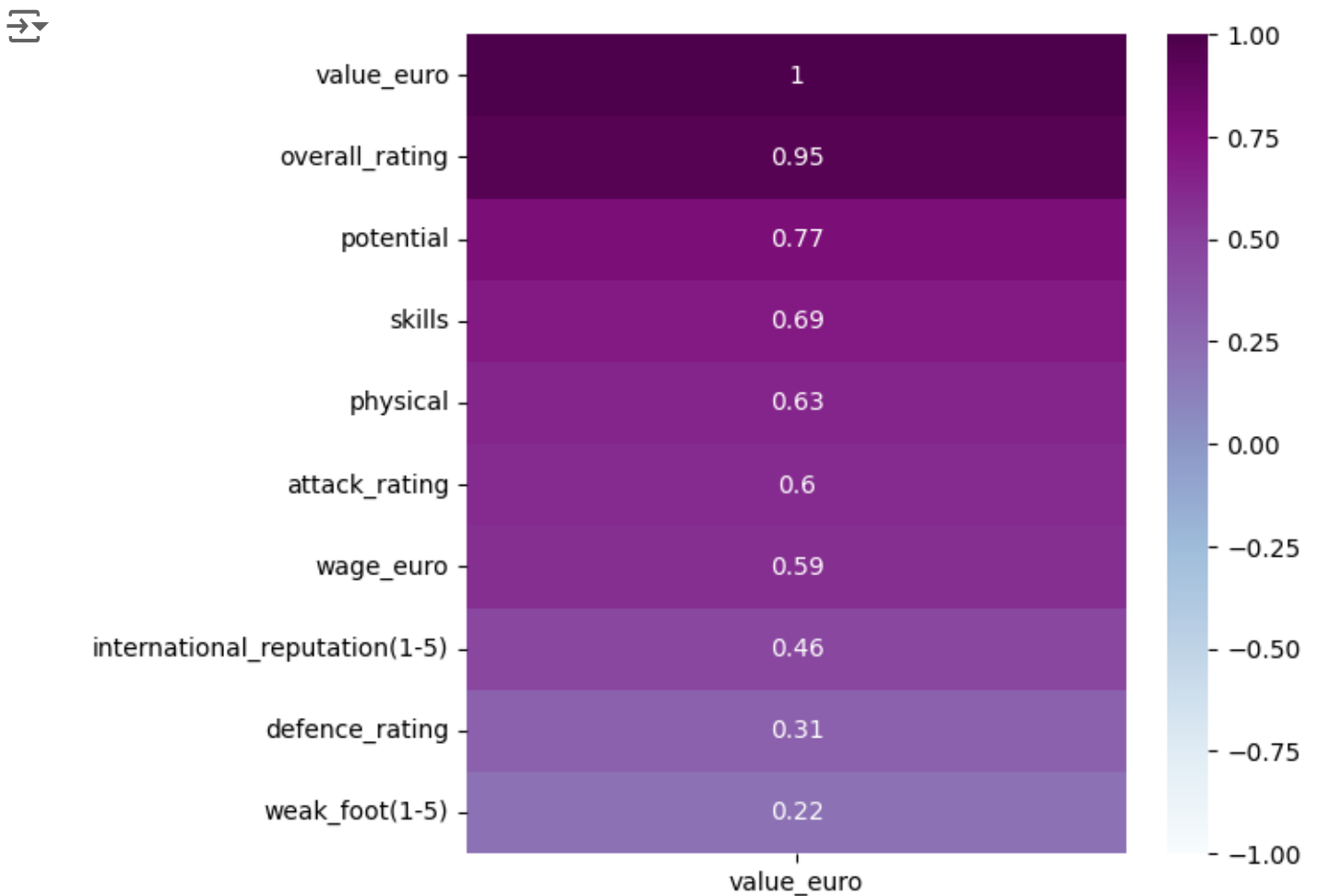
```
corr=df.corr() # gives us the correlation values
plt.figure(figsize=(15,6))
sns.heatmap(corr, annot = True, cmap="BuPu") # let's visualise the correlation
plt.show()
```




```
corr = df.corr()

# Isolate the 'value_euro' column
value_euro_corr = corr[['value_euro']]

# Plot the correlations with 'value_euro'
plt.figure(figsize=(6, 6))
sns.heatmap(value_euro_corr.sort_values(by='value_euro', ascending=False),
            annot=True,
            cmap="BuPu",
            cbar=True,
            vmin=-1,
            vmax=1)
plt.show()
```



Next, count the null/NaN/missing values for each variable

```
df.isnull().sum()
```

```
⇒ overall_rating      0
   potential          0
   value_euro        214
   wage_euro         209
   international_reputation(1-5)  0
   weak_foot(1-5)     0
   physical           0
   defence_rating     0
   skills             0
   attack_rating      0
   dtype: int64
```

Next we replace these missing values, with the mean of the whole column

```
# Fill missing values with the mean
df['value_euro'].fillna(df['value_euro'].mean(), inplace=True)
df['wage_euro'].fillna(df['wage_euro'].mean(), inplace=True)


# Verify that the null values have been filled
df.isnull().sum()
```

```
⇒ overall_rating      0
   potential          0
   value_euro          0
   wage_euro           0
   international_reputation(1-5)  0
   weak_foot(1-5)     0
   physical            0
   defence_rating     0
   skills             0
   attack_rating      0
   dtype: int64
```

```
df.shape
```

```
⇒ (15889, 10)
```

```
df.describe() # Let's view the description of the numerical values in the data
```



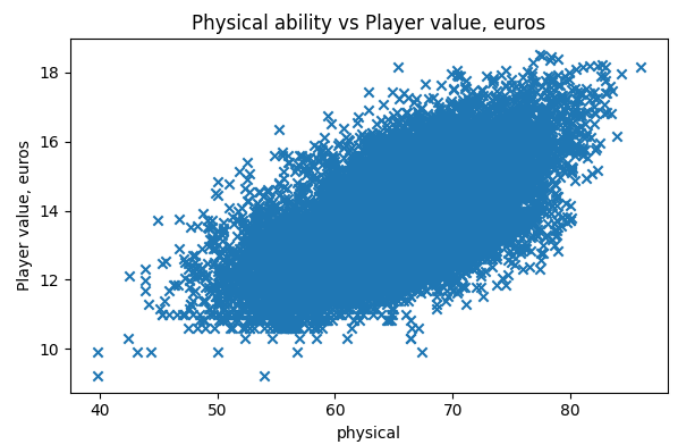
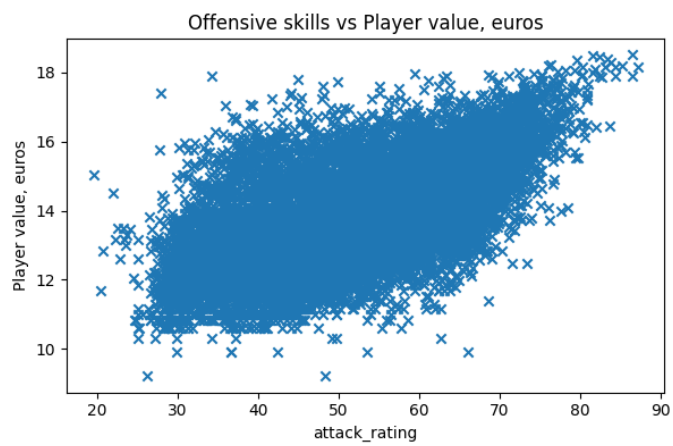
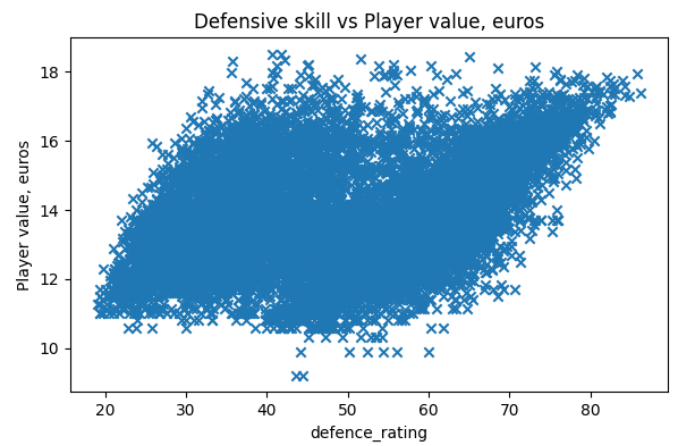
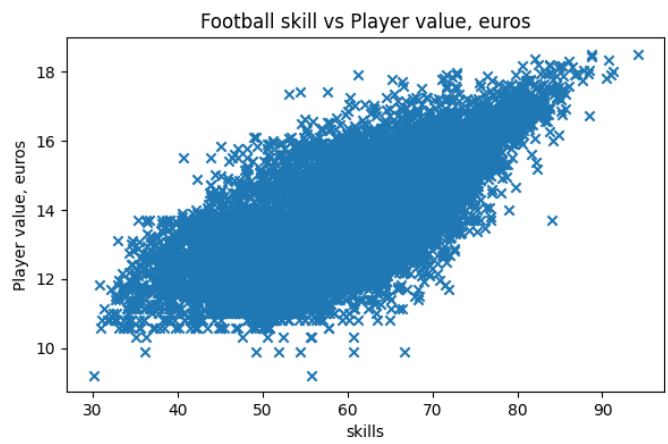
	overall_rating	potential	value_euro	wage_euro	international_r
count	15889.000000	15889.000000	15889.000000	15889.000000	
mean	66.467116	71.639184	13.718492	10313.392857	
std	6.833608	6.073455	1.367575	22423.768685	
min	47.000000	51.000000	9.210340	1000.000000	
25%	62.000000	67.000000	12.834681	2000.000000	
50%	66.000000	71.000000	13.560618	3000.000000	
75%	71.000000	76.000000	14.648420	10000.000000	
max	94.000000	95.000000	18.520526	565000.000000	

Below, we are identifying the correlation between the new features and the target feature. Overall, these scatter plots visually suggest that there are positive correlations between a player's skill ratings in various aspects of the game and their market value. The data points are spread out, indicating variability in how much each skill attribute contributes to a player's value. This kind of visualization is helpful for quickly identifying trends and relationships in the data, which can be useful for further statistical analysis or for informing decisions in player recruitment and valuation.

```
# Section 4: Visualizing Relationships
figure, axis = plt.subplots(2, 2, figsize=(12, 8))
skills = ['skills', 'defence_rating', 'attack_rating', 'physical']
titles = ['Football skill vs Player value, euros', 'Defensive skill vs Player v
        'Offensive skills vs Player value, euros', 'Physical ability vs Playe

for i, ax in enumerate(axis.flat):
    ax.scatter(df[skills[i]], df["value_euro"], marker='x')
    ax.set_title(titles[i])
    ax.set_xlabel(skills[i]) # Set the label for the x-axis
    ax.set_ylabel('Player value, euros') # Set the label for the y-axis
```

```
plt.tight_layout()  
plt.show()
```



✓ 3. Applying the Model

First, we have to split the data into the training set and test set. The model is built around the training data, and is then tested, by comparing the model produced by the training data, to the new unseen data from the test set. We assign the features to X and the target variable (value_euro) to y. We specify that 20% of the data is in the test set.

```
X = df.drop('value_euro', axis=1)
y = df['value_euro']

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply PCA
pca = PCA(n_components=0.95) # Adjust n_components as needed to retain 95% of
X_pca = pca.fit_transform(X_scaled)

# Print the number of components
print("Original feature set size:", X.shape[1])
print("Reduced feature set size:", X_pca.shape[1])

y = np.log(df['value_euro']) # Applying log transformation to the target variable

# Splitting the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, ra
```

```
➦ Original feature set size: 9
  Reduced feature set size: 7
```

```
X_train.shape # the first 422 samples of the 10 variables
```

```
➦ (12711, 7)
```

```
y_train.shape
```

```
➦ (12711,)
```

```
X_test.shape
```

```
➦ (3178, 7)
```

```
y_test.shape
```

```
(3178,)
```

Now we apply Linear regression to our training data

```
# Assuming 'linreg' is intended to be the instance of LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)
```

```
LinearRegression()
```

The Mean Squared Error is a measure of the average squared difference between the predicted values and the actual values, with lower values indicating better fit. The R-squared is a statistical measure of how close the data are to the fitted regression line, with a higher R-squared value indicating a better fit (a maximum value of 1).

The output below the code block shows the computed values for these metrics:

Training Mean Squared Error: This is very low, indicating the model's predictions are, on average, very close to the actual training data points.

Training R-squared: The value of approximately 0.882 suggests that about 88.2% of the variance in the target variable is explained by the model. This indicates a strong fit to the training data.

Test Mean Squared Error: Similar to the training MSE, the test MSE is also low, which suggests good predictive performance on the test set.

Test R-squared: With a value of approximately 0.8793, the model explains about 87.93% of the variance in the test set, indicating that the model generalizes well to unseen data.

```

# Number of observations and number of features
y_train_pred = lin_reg.predict(X_train)
y_test_pred = lin_reg.predict(X_test)
n_train = X_train.shape[0]
n_test = X_test.shape[0]
p = X_train.shape[1]

# Calculate and print the metrics for the training set
mse_train = mean_squared_error(y_train, y_train_pred)
mae_train = mean_absolute_error(y_train, y_train_pred)
sse_train = np.sum((y_train - y_train_pred) ** 2)
r2_train = r2_score(y_train, y_train_pred)
adjusted_r2_train = 1 - (1-r2_train) * (n_train - 1) / (n_train - p - 1)

print(f"Training MSE: {mse_train}")
print(f"Training MAE: {mae_train}")
print(f"Training SSE: {sse_train}")
print(f"Training R-squared: {r2_train}")
print(f"Training Adjusted R-squared: {adjusted_r2_train} \n")

# Calculate and print the metrics for the test set
mse_test = mean_squared_error(y_test, y_test_pred)
mae_test = mean_absolute_error(y_test, y_test_pred)
sse_test = np.sum((y_test - y_test_pred) ** 2)
r2_test = r2_score(y_test, y_test_pred)
adjusted_r2_test = 1 - (1-r2_test) * (n_test - 1) / (n_test - p - 1)

print(f"Test MSE: {mse_test}")
print(f"Test MAE: {mae_test}")
print(f"Test SSE: {sse_test}")
print(f"Test R-squared: {r2_test}")
print(f"Test Adjusted R-squared: {adjusted_r2_test}")

```

```

↗ Training MSE: 0.0011473205189600805
Training MAE: 0.025727444051447425
Training SSE: 14.583591116501584
Training R-squared: 0.8820102226715665
Training Adjusted R-squared: 0.8819452042947028

Test MSE: 0.001189938469243701
Test MAE: 0.026150648563655544
Test SSE: 3.7816244552564813
Test R-squared: 0.8795733404946636
Test Adjusted R-squared: 0.879307414117207

```

Cross validation is a robust method, as the dataset is divided into k folds, which in this case is 5. At each fold, the model gets trained on $k-1$ folds, and gets tested on the remaining folds. This process is repeated k times, and each fold is used once. The cross validation score is calculated as the average. In this case, the cross validation provides me with a R-squared score of 0.882. The cross-validation results are very close to the initial train/test split results, which suggests that the model's performance is consistent across different subsets of the data. This consistency is crucial for confirming the model's reliability and generalizability. The slight variations in metrics before and after cross-validation (such as MSE and MAE) are expected due to the variability inherent in using different subsets of data for training and validation.

```
def train_and_evaluate(model, X, y):
    """
    Train and evaluate a regression model using both a single train/test split
    and cross-validation to compare performances, including evaluations on
    both training and test sets for R^2, MSE, and MAE.
    """
    print('Evaluating model:', model.__class__.__name__)

    # Perform a train/test split
    X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2)

    # Cross-validation for model evaluation
    cv_scores_r2 = cross_val_score(model, X_train, y_train, cv=5, scoring='r2')
    cv_scores_mse = cross_val_score(model, X_train, y_train, cv=5, scoring='neg
    cv_scores_mae = cross_val_score(model, X_train, y_train, cv=5, scoring='neg

    print('Cross-Validation Mean R^2:', np.mean(cv_scores_r2), 'with SD:', np.s
    print('Cross-Validation Mean MSE:', -np.mean(cv_scores_mse), 'with SD:', np
    print('Cross-Validation Mean MAE:', -np.mean(cv_scores_mae), 'with SD:', np

# Example usage of the function with Linear Regression
train_and_evaluate(LinearRegression(), X_pca, y)
```

```
➡ Evaluating model: LinearRegression
Cross-Validation Mean R^2: 0.8816318206500029 with SD: 0.005907878794424556
Cross-Validation Mean MSE: 0.0011495545110000106 with SD: 4.368224802123239
Cross-Validation Mean MAE: 0.025746147741548042 with SD: 0.0003802492708855
```


Regularisation is a technique used that prevents overfitting (when a model performs badly with unseen data). It occurs when a model is too complex or when it has many parameters. Regularisation addresses this by adding a penalty, where the model is minimising the error between predicted and actual values (MSE) but also to keep the model coefficients small.

2 techniques used in this notebook are Lasso and ridge. Here we use LassoCV and RidgeCV, which uses cross validation to determine the best value for alpha, which for Lasso is the magnitude of the penalty term which is the absolute value of the magnitude of the coefficients and for ridge it is proportional to the square of the coefficient values.

```
X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, ra
```

```
# Initialize and fit the LassoCV model
lasso_cv = LassoCV(alphas=None, cv=10, max_iter=10000)
lasso_cv.fit(X_train, y_train)

# Predictions
y_train_pred_lasso_cv = lasso_cv.predict(X_train)
y_test_pred_lasso_cv = lasso_cv.predict(X_test)

# Optimal alpha
print("Optimal alpha for LassoCV: ", lasso_cv.alpha_)

# R^2 scores
lasso_cv_train_score = lasso_cv.score(X_train, y_train)
lasso_cv_test_score = lasso_cv.score(X_test, y_test)

print(f"LassoCV Training Score (R^2): {lasso_cv_train_score}")
print(f"LassoCV Test Score (R^2): {lasso_cv_test_score}")

# MSE calculations
mse_train = mean_squared_error(y_train, y_train_pred_lasso_cv)
mse_test = mean_squared_error(y_test, y_test_pred_lasso_cv)

print(f"LassoCV Training MSE: {mse_train}")
print(f"LassoCV Test MSE: {mse_test}")
```

```
➞ Optimal alpha for LassoCV: 0.0001779832737187039
LassoCV Training Score (R^2): 0.8819763956390468
LassoCV Test Score (R^2): 0.8795837952172142
LassoCV Training MSE: 0.0011476494495622405
LassoCV Test MSE: 0.0011898351659004106
```

```
# Define a set of alphas to consider
alphas = np.logspace(-6, 6, 13)

# Initialize and fit the RidgeCV model
ridge_cv = RidgeCV(alphas=alphas, cv=10)
ridge_cv.fit(X_train, y_train)

# Predictions for training and testing sets
y_train_pred_ridge_cv = ridge_cv.predict(X_train)
y_test_pred_ridge_cv = ridge_cv.predict(X_test)

# Optimal alpha value
print("Optimal alpha for RidgeCV: ", ridge_cv.alpha_)

# R^2 scores for training and testing sets
ridge_cv_train_score = ridge_cv.score(X_train, y_train)
ridge_cv_test_score = ridge_cv.score(X_test, y_test)
print(f"RidgeCV Training Score (R^2): {ridge_cv_train_score}")
print(f"RidgeCV Test Score (R^2): {ridge_cv_test_score}")

# Calculating and printing the MSE for the training and testing sets
mse_train = mean_squared_error(y_train, y_train_pred_ridge_cv)
mse_test = mean_squared_error(y_test, y_test_pred_ridge_cv)
print(f"RidgeCV Training MSE: {mse_train}")
print(f"RidgeCV Test MSE: {mse_test}")
```

```
➞ Optimal alpha for RidgeCV: 10.0
RidgeCV Training Score (R^2): 0.8820100676153063
RidgeCV Test Score (R^2): 0.8795778196452243
RidgeCV Training MSE: 0.0011473220267112856
RidgeCV Test MSE: 0.0011898942106585702
```

Now, I would like to see visually, how the model fits onto the input data. The following code plots the actual players's values against their predicted values, as well as plotting the model onto the data. If in the plots, the datapoints are closer to the dashed line, it shows that the predictions are accurate, as it shows that the prediction values are close to the actual values.

The LassoCV plot utilizes Lasso regression with cross-validation to optimize the alpha parameter, which is the regularization strength. Lasso regression is a type of linear regression that adds a regularization term to the cost function. The regularization term penalizes large coefficients; thus, it can set some coefficients to zero, effectively performing feature selection.

Similar to LassoCV, RidgeCV applies Ridge regression with cross-validation. Ridge regression also adds a regularization term.

It can be seen in the plots that the LassoCV and the RidgeCV and simple Linear Regression Plot all perform similarly.

```
plt.figure(figsize=(8,15))

# Linear Regression plot
plt.subplot(3, 1, 1)
plt.scatter(y_test, y_test_pred, alpha=0.5)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--', lw=
plt.title('Actual vs. Predicted Player Values (Test Set)')
plt.xlabel('Actual Values (Euro)')
plt.ylabel('Predicted Values (Euro)')

# LassoCV plot - regularisation 1
plt.subplot(3, 1, 2)
plt.scatter(y_test, y_test_pred_lasso_cv, alpha=0.5)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--', lw=
plt.title('LassoCV\nActual vs. Predicted Player Values')
plt.xlabel('Actual Values (Euro)')
plt.ylabel('Predicted Values (Euro)')

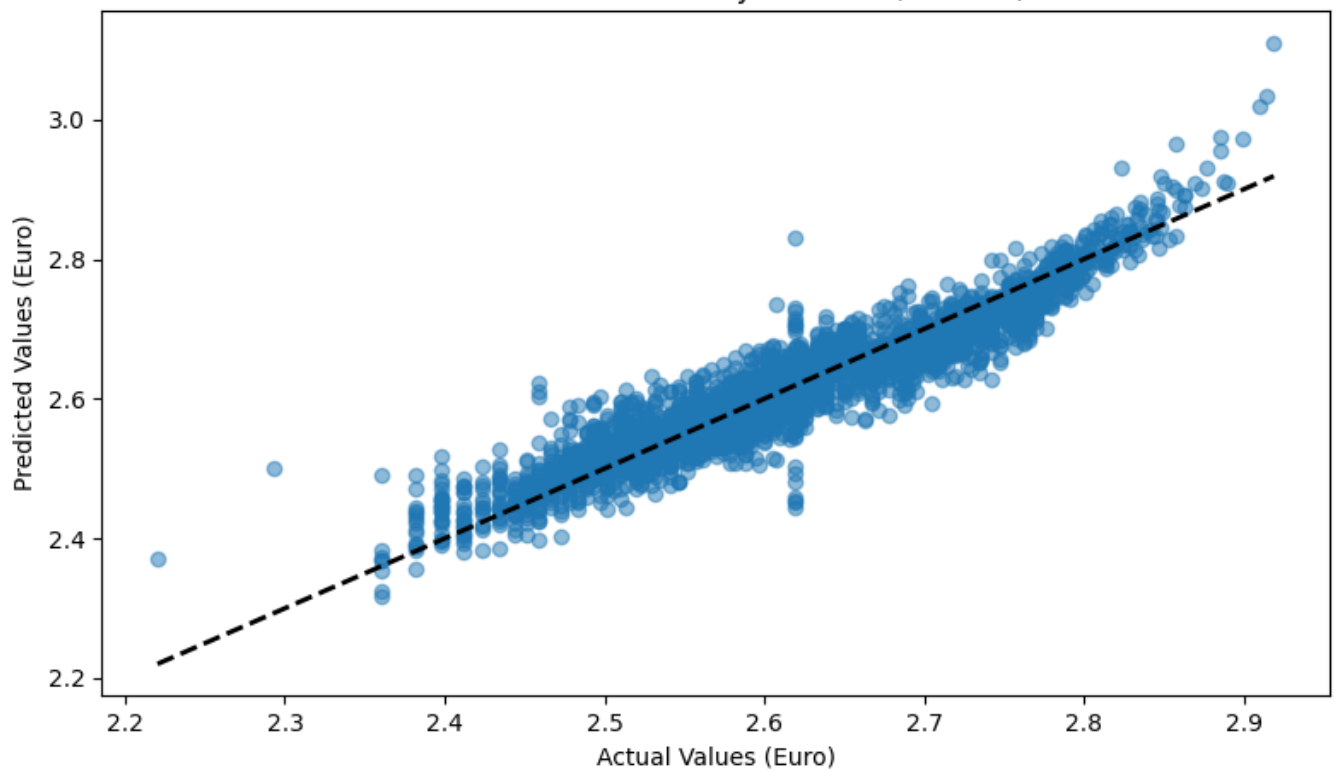
# RidgeCV plot - regularisation 2
plt.subplot(3, 1, 3)
plt.scatter(y_test, y_test_pred_ridge_cv, alpha=0.5)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--', lw=
plt.title('RidgeCV\nActual vs. Predicted Player Values')
plt.xlabel('Actual Values (Euro)')
plt.ylabel('Predicted Values (Euro)')

plt.tight_layout()
```

```
plt.show()
```

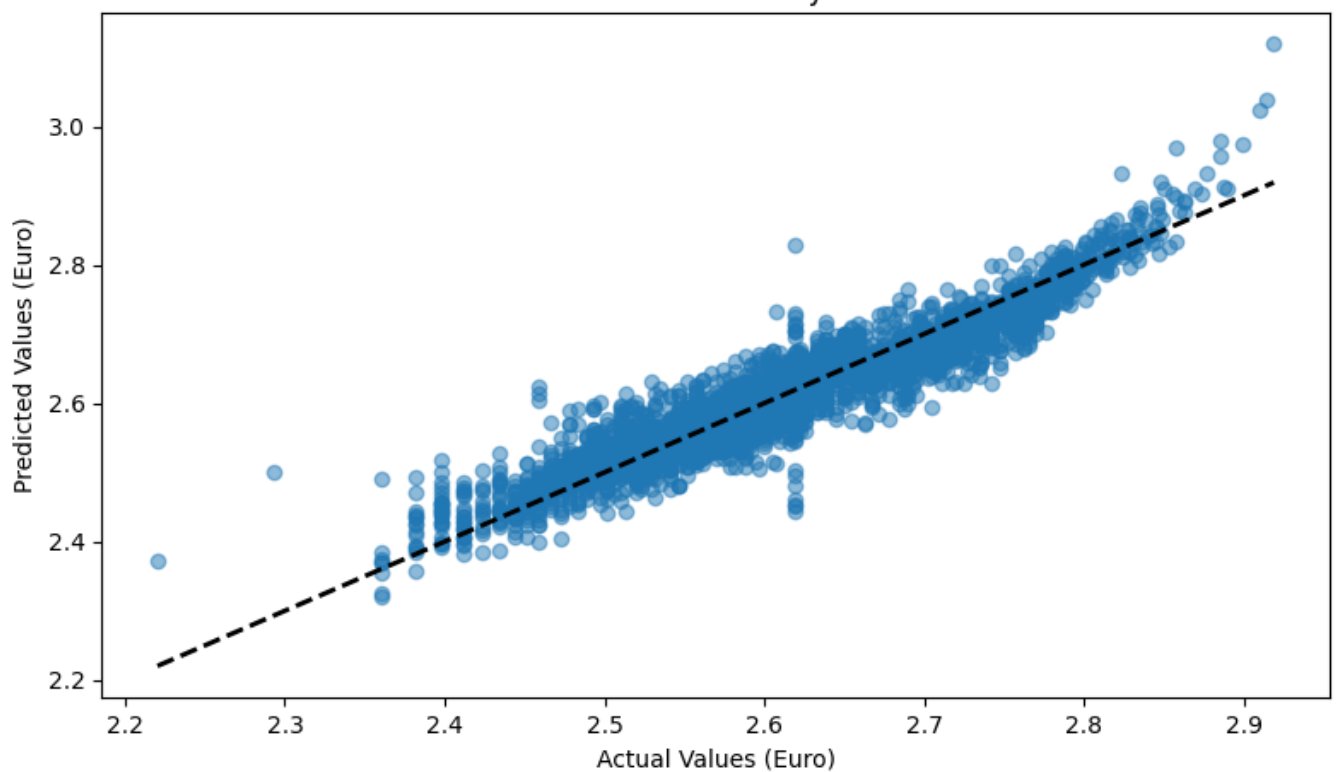


Actual vs. Predicted Player Values (Test Set)



LassoCV

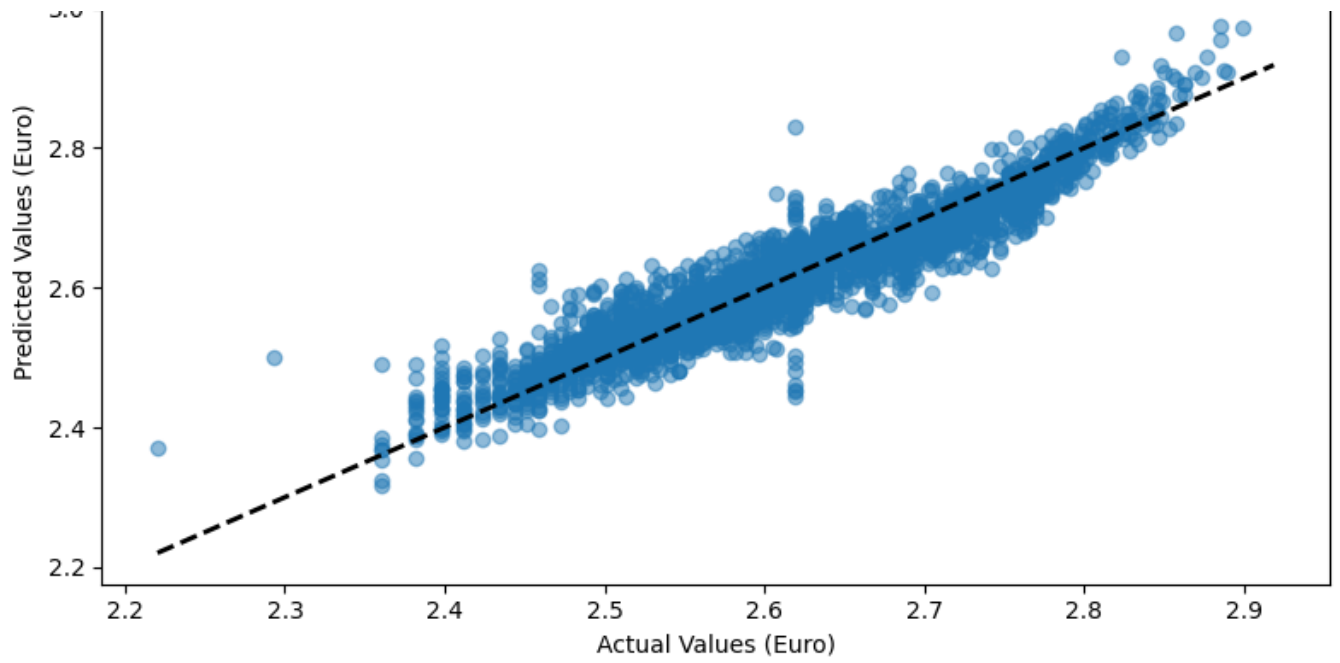
Actual vs. Predicted Player Values



RidgeCV

Actual vs. Predicted Player Values





Now we will try another model technique on our data to predict the value of a player. This time we will use Decision Trees. Decision trees predicts the value of a target variable by learning simple decision rules inferred from the data features. In this regression problem, the decision at each node splits the data such that the reduction in variance of the target variable is maximized. The tree is constructed by splitting the dataset into subsets based on an attribute value test. This process is recursively repeated on each derived subset.

To avoid overfitting, we need to determine the optimal value for the max_depth of the tree, as well as the minimum amount of sample. To determine the optimal value for each of these variables, we perform cross validation using GridSearchCV. This object works through multiple combinations of parameter values, cross-validating as it goes to determine which tune gives the best performance. The output of GridSearchCV will give you the best parameter set found during the grid search.

```
# Split your data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, ra

# Define the parameter grid to search
param_grid = {
    'max_depth': range(1, 20),
    'min_samples_leaf': range(1, 20)
}

# Instantiate the regressor
dt_reg = DecisionTreeRegressor(random_state=0)

# Create the GridSearchCV object
grid_search = GridSearchCV(estimator=dt_reg, param_grid=param_grid, cv=10, scor

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Print the best parameters found
print(f"Best parameters: {grid_search.best_params_}")
```

➡ Best parameters: {'max_depth': 12, 'min_samples_leaf': 18}

Now, we apply the Best parameter values obtained above to the decision tree, and using it to predict the transfer value of the players. The output then shows the R-squared score and the MSE for both the training set and the test set. The R-squared value is 0.9432 for the training set, and 0.9161 for the test set, and also the MSE value is very low for both (0.00055234 for training set, 0.0008290 for test set). The model appears to be performing well, given the high R-squared values and low MSE on both the training and test sets. The fact that the R-squared value and MSE have not degraded much from training to testing after performing a grid search and cross-validation indicates that the grid search for the hyperparameters has helped the model to predict the target more accurately

```
# Initializing the Decision Tree Regressor with a random state for reproducibil
decision_tree = DecisionTreeRegressor(
    max_depth=grid_search.best_params_['max_depth'],
    min_samples_leaf=grid_search.best_params_['min_samples_leaf'],
    random_state=0
)

# Training the model on the training set
decision_tree.fit(X_train, y_train)

# Predicting the target variable for the training and test sets
y_train_pred = decision_tree.predict(X_train)
y_test_pred = decision_tree.predict(X_test)

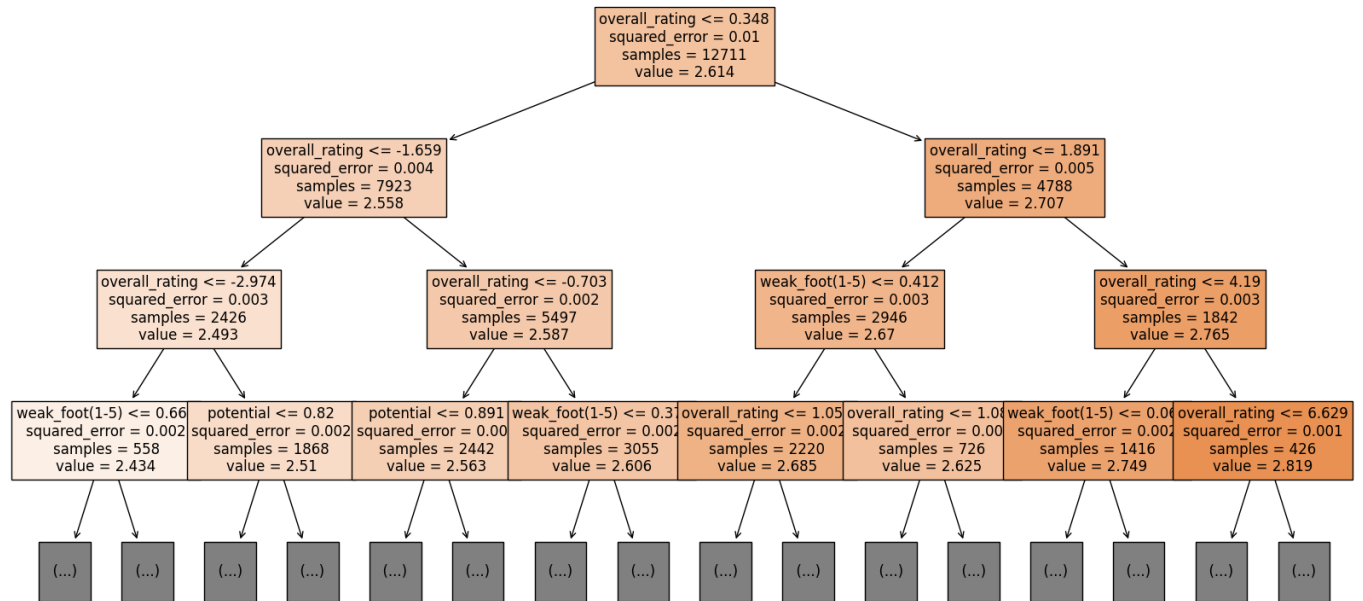
# Calculating the R-squared value and Mean Squared Error for both training and
train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)
train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)

print(f"Training R-squared (Accuracy): {train_r2}")
print(f"Test R-squared (Accuracy): {test_r2}")
print(f"Training Mean Squared Error: {train_mse}")
print(f"Test Mean Squared Error: {test_mse}")

# Plotting the tree (with limited depth for readability)
plt.figure(figsize=(20,10))
plot_tree(decision_tree, max_depth=3, feature_names=X.columns, filled=True, for
plt.show()
```



Training R-squared (Accuracy): 0.9431978188167434
Test R-squared (Accuracy): 0.9160991982603943
Training Mean Squared Error: 0.0005523385963500209
Test Mean Squared Error: 0.0008290256659151254



✓ Random Forest Regressor

Finally, we apply another type of decision tree model called the Random Forest Regressor to the data, and we again use GridCV to choose the best parameters for the model. It can be seen that this model has performed the best over both training accuracy and cross validation accuracy

```
# Assuming 'X_pca' and 'y' are your PCA-transformed features and target variable
X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, ran

# Define the parameter grid to search
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10]
}

# Instantiate the regressor
rf_reg = RandomForestRegressor(random_state=42)

# Create the GridSearchCV object
grid_search = GridSearchCV(estimator=rf_reg, param_grid=param_grid, cv=10, scori

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Print the best parameters found
print(f"Best parameters: {grid_search.best_params}")
```

➡ Fitting 10 folds for each of 36 candidates, totalling 360 fits
Best parameters: {'max_depth': 20, 'min_samples_split': 5, 'n_estimators':

```
# Initializing the RandomForestRegressor with best parameters found
random_forest = RandomForestRegressor(
    n_estimators=grid_search.best_params_['n_estimators'],
    max_depth=grid_search.best_params_['max_depth'],
    min_samples_split=grid_search.best_params_['min_samples_split'],
    random_state=42
)

# Training the model on the training set
random_forest.fit(X_train, y_train)

# Predicting the target variable for the training and test sets
y_train_pred = random_forest.predict(X_train)
y_test_pred = random_forest.predict(X_test)

# Calculating the R-squared value and Mean Squared Error for both training and test sets
train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)
train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)

print(f"Training R-squared (Accuracy): {train_r2}")
print(f"Test R-squared (Accuracy): {test_r2}")
print(f"Training Mean Squared Error: {train_mse}")
print(f"Test Mean Squared Error: {test_mse}")
```

↗ Training R-squared (Accuracy): 0.9871647308670015
Test R-squared (Accuracy): 0.9375124789485147
Training Mean Squared Error: 0.00012480884341083817
Test Mean Squared Error: 0.0006174405688263989

