PREDICTING SOCCER PLAYERS VALUE WITH FIFA RATINGS

By Christian Basinger, Joyal Joby Chully, Firdhous Puraliyil, Shubham, Yiwei Zhao

Business Problem:

Our group wanted to solve a problem that interested our hobbies. We soon found out that we all liked the sport soccer and bonded over that. We then realized we also liked the video game FIFA. Upon talking about the video game, we were curious about the franchise mode. We realized when you are a manager of a team, and you want to acquire a player from another team, you have to pay a certain fee to acquire said player. However, the fee isn't a one to one with the player ratings, and we soon realized that we could possibly make a model to figure out how much a player's value is based on the player's attributes. Our group then also realized how this could be an actual real-life business problem. What if we could help clubs evaluate players' worth by the rating that FIFA provides so that they don't overpay for a player or undersell a player on their club. This way clubs can save money and make smart decisions.

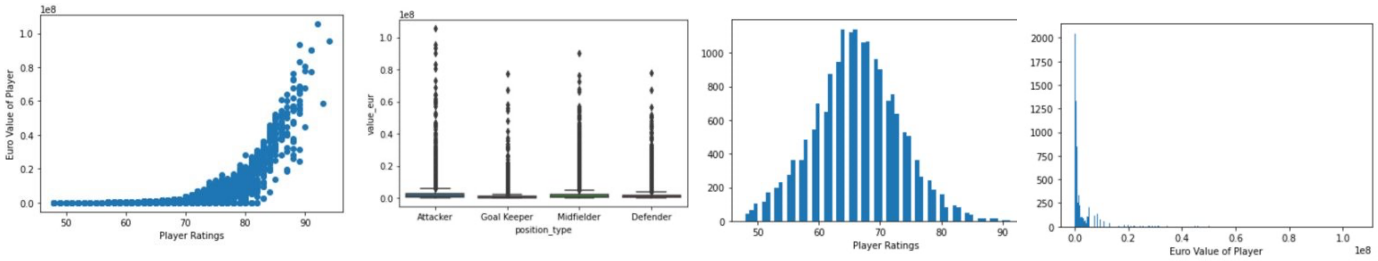
Data:

The data was obtained from kaggle and is the data from FIFA 20. This data set has 16780 records and 78 attributes. The attributes range from string variables to numeric variables. The numeric variables range from 0 to 99 except for the wage and player value variables.

Data Link: https://www.kaggle.com/datasets/stefanoleone992/fifa-20-complete-player-dataset?select=players_20.csv

Visualization:

After composing a scatter plot between overall rating and player value, we can see that it is not a one to one comparison. In fact the graph is an exponential graph, showing that the more the player is rated, the higher the value.



The player rating distribution is normally distributed, however, the player value distribution is skewed right, showing that most players make salaries well below the mean, proving again that a player value is not one to one with overall rating.

But do all positions have the same salaries compared to their rating? By doing a box plot amongst the four player types, Attacker (players who score), Midfielder (player swho scores and defends), Defender (Player who defends attackers) and Goalkeeper, we can see that the IQR are different for each player type and that there are more outliers for attackers. After doing a summary on each group we can also infer that each attribute has a different effect on each player type rating's. For example, while the mean overall rating for attacker and defender are similar, the mean for shooting is drastically different for attackers and defenders. In the next section, we will talk about how we proceeded to compensate for this issue.

	overall								shooting							
	count	mean	std	min	25%	50%	75%	max	count	mean	std	min	25%	50%	75%	max
position_type																
Attacker	3442	66.65078	7.109887	48	62	66	71	94	3442	64.49245	7.773686	36	59	64	70	93
Defender	5938	66.28259	6.548813	48	62	66	70	90	5938	40.06383	11.41561	15	31	38	48	77
Goal Keeper	2036	64.79666	7.603627	48	60	65	70	91	0							
Midfielder	6862	66.43865	6.950925	48	62	66	71	91	6862	56.76873	9.935193	24	50	57	64	88

	value_eur						
	count	mean	std	min	25%	50%	75%
position_type							
Attacker	3442	3164576	7346654	0	400000	875000	2800000
Defender	5938	2094454	4475174	0	325000	675000	1800000
Goal Keeper	2036	1728016	4973524	0	140000	400000	975000
Midfielder	6862	2704119	5548817	0	375000	800000	2400000

Data Cleaning & Feature Engineering:

We think Score players play for different positions should use different attributes to determine their values. For instance, Middlefield player who has better passing skills may be worth more value; good defenders should be focus on tackling skills; and for the attackers the shooting skills probably is the most important attribute to determine their values.

In order to make the model more accurate, in this project we will only focus on the attackers. We select ST, CF, LW, RW those four positions as attackers. If you look at the player_position column from dataset you can see some of the players can play more than one positions,

	short_name	player_positions
0	L. Messi	RW, CF, ST
1	Cristiano Ronaldo	ST, LW
2	Neymar Jr	LW, CAM
3	J. Oblak	GK
4	E. Hazard	LW, CF

therefore, we separate the player_position column by comma and create another column called attackers and integrate ST, CF, LW, RW those positions as attackers

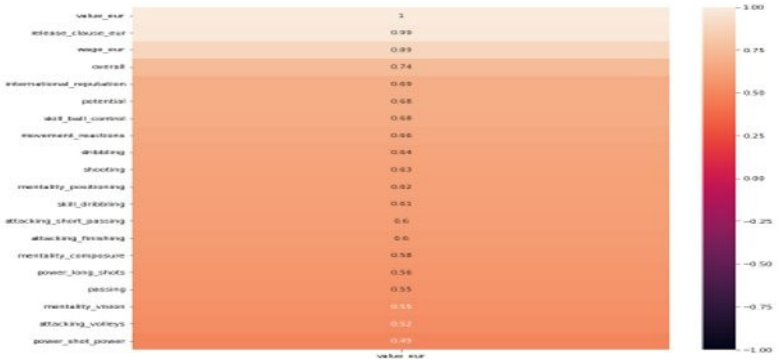
	Position_CAM	Position_CB	Position_CDM	Position_CF	Position_CM	Position_GK	Position_LB	Position_LM	Position_LW	Position_LWB	Position_RB
0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0
2	1	0	0	0	0	0	0	0	1	0	0
3	0	0	0	0	0	1	0	0	0	0	0
4	0	0	0	1	0	0	0	0	1	0	0

```
fifa['attacker'] = fifa['Position_CF'] + fifa['Position_LW'] + fifa['Position_RW'] + fifa['Position_ST']
```

and then we filling out the NA value with median and scaling the data with standardscaller.

```
for column in columns:
    fifa[column] = fifa[column].fillna(fifa[column].median())
fifa[columns]
```

After that, we dropped all the columns separated from player_position column including player_position column as well,



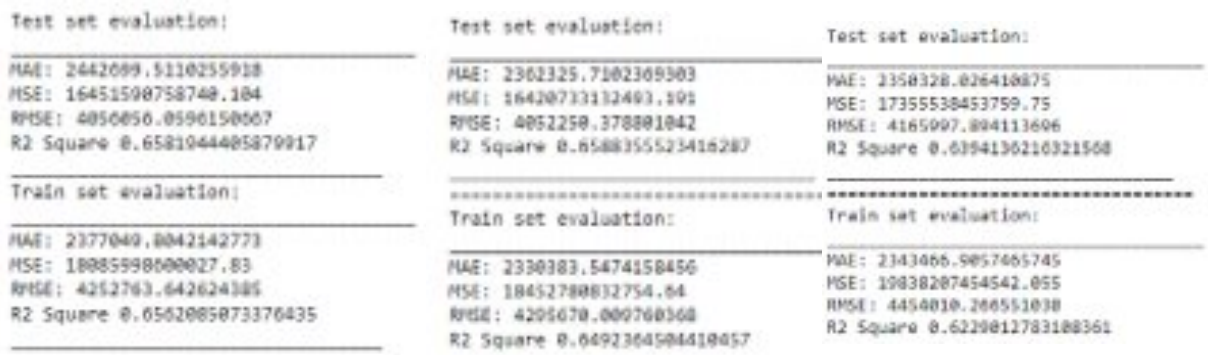
and run the correlation heatmap to selected the top 45 correlated numeric attributes, make sure that dataset is ready to do the further model selection.

Models:

Regression model

Regression model is a function that describes the relationship between one or more independent variables and a dependent or target variable. In order to discuss which variable is related to

player's value we will run linear, ridge and lasso regression in this project. We split the dataset with 70% as training and 30% as test. We got 0.6581 R^2 for Linear regression, 0.6588 R^2 for Ridge regression and 0.6394 R^2 for Lasso regression.



SVM

We used the the SVR model since we are using the model for a regression. First, we used a standard scaler to scale the data since svm uses a distance measure for its predictions. Then we applied a grid search to try to find the best hyperparameters. We used Cs from .01 up to 100000, gammas from .001 up to 100, and both linear and rbf kernels. Initially with such a large set a hyperparameters to use, we used a subset of the train data, 25% random sample, to tune the model. Since svm is a computationally heavy model, this helped speed it up and gave us a smaller set of parameters to choose from. Once we had a smaller set of hyperparameters to choose from, we ran it with the whole training data set.

The hyperparameters that were chosen was C = 100000 and Linear as the kernel. The test accuracy was just over 48%. Some potential ways to improve this model would be to drop some of the variables and focus on a handful that have the most impact on the target's variance. When experimenting with this using the same chosen model, we had accuracy near 60% and that was without running another grid search for additional tuning. However, we wanted to be consistent with the variables used for all the models which is why we didn't pursue additional tuning with less variables.

```
from sklearn.svm import SVR
from sklearn.model_selection import GridSearchCV

param_grid = {'C':  [1000, 10000, 100000],
              'gamma':  [.01, .001, .0001],
              'kernel': ['rbf', 'linear']
}

model = SVR()

grid_svr = GridSearchCV(model, param_grid, n_jobs = 2)

grid_svr.fit(X_train, y_train)
```

```
Test set evaluation:
_____________________________________
MAE: 1899852.0560912145
MSE: 24978599597472.23
RMSE: 4997859.501573872
R2 Square 0.48103351621442814
```

```
print(f'The best estimator is:{grid_svr.best_estimator_}\nThe best score is:{grid_svr.score(X_test, y_test)}')
```

```
The best estimator is:SVR(C=100000, gamma=0.0001, kernel='linear')
The best score is:0.48103351621442814
```

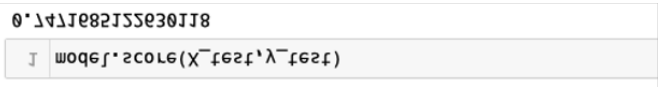
Knn regressor

We implemented a knn regressor model for prediction. Towards this, first step that we had to do was to scale the data using MinMaxScaler and brought the data values between zero and one since, knn is dependent on distance attribute. Then to optimize the number of neighbors, a 5-fold cross validation was performed starting with a base knn model. The tuning range was given as odd numbers from 1(inclusive) to 26.

```
# Gridsearch
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsRegressor

# define function
knn = KNeighborsRegressor()
params = {'n_neighbors':list(range(1,26,2))}


model = GridSearchCV(knn, params, cv=5)
model.fit(X_train,y_train)
print("Best k is:",model.best_params_)
print("Mean Validation score is:",model.best_score_)
```

```
Best k is: {'n_neighbors': 11}
Mean Validation score is: 0.7700227199068768
```

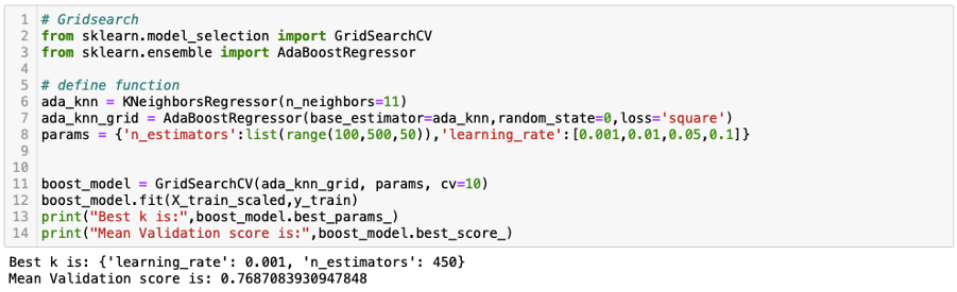
Through cross-validation, we found that the optimum number of neighbors were 11 with score of 0.77. We evaluated the model with test data.



```
MAE: 6012.890598636707
MSE: 189841478.1546996
RMSE: 13778.297360512277
```

We got a score of 0.747 for the test set, Mean Absolute error of 6012.89, Mean Square Error of 189841478.15 and Root Mean Square Error of 13778.29.

Furthermore, we decided to boost this model to check if this model would show a better performance with Adaboost and cross-validation.



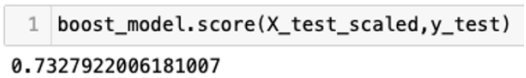
```
# Gridsearch
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import AdaBoostRegressor

# define function
ada_knn = KNeighborsRegressor(n_neighbors=11)
ada_knn_grid = AdaBoostRegressor(base_estimator=ada_knn,random_state=0,loss='square')
params = {'n_estimators':list(range(100,500,50)),'learning_rate':[0.001,0.01,0.05,0.1]}


boost_model = GridSearchCV(ada_knn_grid, params, cv=10)
boost_model.fit(X_train_scaled,y_train)
print("Best k is:",boost_model.best_params_)
print("Mean Validation score is:",boost_model.best_score_)
```

```
Best k is: {'learning_rate': 0.001, 'n_estimators': 450}
Mean Validation score is: 0.7687083930947848
```

We used the knn model with 11 neighbors as the base estimator and square loss function. Parameters for estimation were n_estimators and learning rate. n_estimator range was given from 100 to 500 with a step of 50 and learning rates were 0.001, 0.01, 0.05 and 0.1.

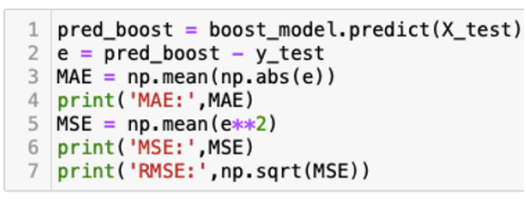
We performed a 10 fold cross-validation. The optimum learning rate we got was 0.01 and optimum number of estimators was 450 with a validation score of 0.768.

We evaluated the chosen model with test.



```
boost_model.score(X_test_scaled,y_test)
```

```
0.7327922006181007
```



```
pred_boost = boost_model.predict(X_test)
e = pred_boost - y_test
MAE = np.mean(np.abs(e))
print('MAE:',MAE)
MSE = np.mean(e**2)
print('MSE:',MSE)
print('RMSE:',np.sqrt(MSE))
```

```
MAE: 5903.016805021267
MSE: 179378938.19752523
RMSE: 13393.242258599119
```

We got a testing accuracy of 0.7327. Mean absolute error was 5903, Mean Squared error was 179378938.19 and Root Mean Squared error was 13393.24.

The evaluation for basic knn-regressor and boosted knn-regressor were very much close but, the runtime for boosted knn was very high comparatively.

Decision Tree

We implemented Decision Tree for prediction of salary of FIFA players. We had already trained the dataset, and implemented the algorithm using scikit-learn package. We imported the DecisionTreeRegressor from sklearn.tree library and implemented the model. Then, we fit the model with the training set with .fit() method. Then, we looked at accuracy for training and test set.

The decision tree gave an accuracy of 0.925 for the test set and 1 for the training set, indicating overfitting, which means the model's prediction is more aligned to the training set.

```
from sklearn.tree import DecisionTreeRegressor

tree_complete = DecisionTreeRegressor(random_state = 0)

tree_complete.fit(X_train, y_train)
```

```
DecisionTreeRegressor(random_state=0)
```

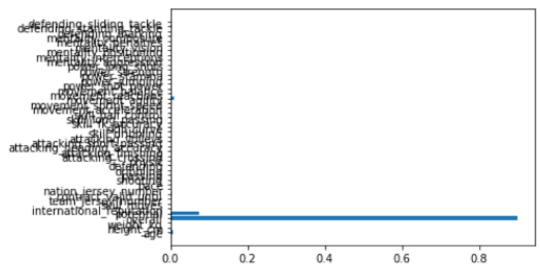
```
tree_complete.score(X_test, y_test), tree_complete.score(X_train, y_train)
```

```
(0.9254049661336998, 1.0)
```

We also looked at the feature importance using feature_importances_,

```
x = var
y = tree_complete.feature_importances_
plt.barh(x, y)
plt.figure(figsize=(20,20))
```

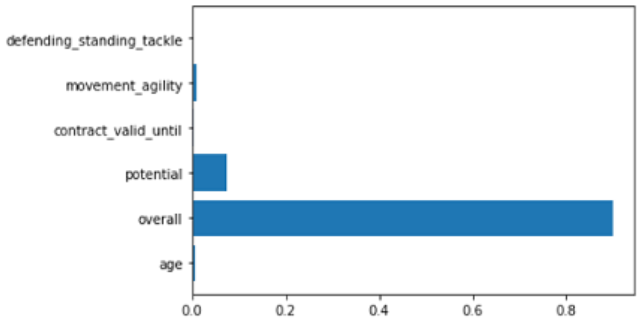
```
<Figure size 1440x1440 with 0 Axes>
```



To get better idea about the main features, we looked at features having positive value greater than 0.002

```
my_idx = tree_complete.feature_importances_ > 0.002
x = np.array(var)[my_idx]
y = tree_complete.feature_importances_[my_idx]
plt.barh(x, y)
```

We observed few features like “defending standing tackle”, “movement agility”, “potential” and others. We observed the feature 'overall' as the most important feature.



To optimize the model and overcome overfitting, hyperparameter tuning was performed using GridSearchCV.

We observed best model’s hyperparameters as:

Maximum Depth as 5

Maximum leaf node as 10

Minimum samples split as 8

```
#Hyper-parameter Tuning for Decision Tree
# GridSearch + CV
from sklearn.model_selection import GridSearchCV

opt_tree = DecisionTreeRegressor(random_state = 0)
dt_params = {'max_depth':  range(1,10)            ,
             'min_samples_split':   range(2,11),
             'max_leaf_nodes':    range(2,11)   }

# 5 for depth, 8 for sample, 10 for Leaf
grid_tree = GridSearchCV(opt_tree, dt_params)
grid_tree.fit(X_train, y_train)
```

```
5]: GridSearchCV(estimator=DecisionTreeRegressor(random_state=0),
                 param_grid={'max_depth': range(1, 10),
                             'max_leaf_nodes': range(2, 11),
                             'min_samples_split': range(2, 11)})
```

```
# Report the best hyperparameters chosen
grid_tree.best_params_
```

```
6]: {'max_depth': 5, 'max_leaf_nodes': 10, 'min_samples_split': 8}
```

After running the model, we observed an accuracy of 0.95 on the training set and 0.91 on test set indicating we have reduced the overfitting

```
print("The accuracy for training set is:",round(tree_complete_best_params.score(X_train, y_train),4))
print("The accuracy for test set is:",round(tree_complete_best_params.score(X_test, y_test),4))
```

```
The accuracy for training set is: 0.9545
The accuracy for test set is: 0.9114
```

Random Forest

We implemented Random Forest algorithm to build the model using sklearn library

We observed an accuracy of 0.797 for the training set and 0.831 for the test set

```
from sklearn.ensemble import RandomForestRegressor

rnd_clf = RandomForestRegressor(n_estimators=200, max_samples=100,max_depth=6, min_samples_split=2,max_leaf_nodes=10, random_
rnd_clf.fit(X_train, y_train)

print("The accuracy for training set is:",round(rnd_clf.score(X_train, y_train),4))
print("The accuracy for test set is:",round(rnd_clf.score(X_test, y_test),4))
```

```
The accuracy for training set is: 0.7978
The accuracy for test set is: 0.831
```

For optimization, we implemented GridSearchCV for obtaining best model's hyperparameter

Best Hyperparameter and Accuracy:

- Max depth:6
- Max leaf nodes: 10
- Min samples split: 6
- Accuracy for train set: 0.9652

- Accuracy for test set: 0.9438

```
#Hyper-parameter Tuning for random forest
# GridSearch + CV
from sklearn.model_selection import GridSearchCV

hp_rnd_clf = RandomForestRegressor(random_state = 0)

dt_params = {'max_depth':  range(1,10)        ,
             'min_samples_split':   range(2,11),
             'max_leaf_nodes':    range(2,11)   }

# 9 for depth, 10 for sample, 10 for leaf

grid_tree_rf = GridSearchCV(hp_rnd_clf, dt_params)

grid_tree_rf.fit(X_train, y_train)
```

```
GridSearchCV(estimator=RandomForestRegressor(random_state=0),
             param_grid={'max_depth': range(1, 10),
                         'max_leaf_nodes': range(2, 11),
                         'min_samples_split': range(2, 11)})
```

```
grid_tree_rf.best_estimator_
```

```
RandomForestRegressor(max_depth=6, max_leaf_nodes=10, min_samples_split=6,
                      random_state=0)
```

```
print("The accuracy for training set is:",round(rnd_clf_best_param.score(X_train, y_train),4))
print("The accuracy for test set is:",round(rnd_clf_best_param.score(X_test,y_test),4))
```

```
The accuracy for training set is: 0.9652
The accuracy for test set is: 0.9438
```

We also, implemented Adaboost with a maximum depth of 6 and learning rate of 0.5, and observed an accuracy of 0.968 on the test set

```
#AdaBoost
from sklearn.ensemble import AdaBoostRegressor

# Define base model
naive_dt = DecisionTreeRegressor(max_depth=6)

# AdaBoost
ada_clf = AdaBoostRegressor(
    naive_dt, n_estimators = 200, learning_rate=0.5,
    random_state=0)

ada_clf.fit(X_train, y_train)

# Performance
print(ada_clf.score(X_test, y_test))
```

```
0.9723058407057359
```

Model Comparison:

Below are the models used in our analysis and the test accuracy achieved. Random Forest and Decision Tree with Adaboost were by far the most successful in predicting players value compared to the other models.

Model	Test Score
Decision Tree with Adaboost	96.9%
Random Forest	94.4%
Decision Tree	91.1%
KNN with Adaboost	76.1%
KNN	74.7%
Ridge Regression	65.9%
Linear Regression	65.8%
Lasso Regression	63.9%
SVM	48.1%

Conclusion:

Overall, we were able to build a model to predict players value using their FIFA rating. The Decision Tree with AdaBoost and Random Forest models did far better than the other ones we used with Test accuracy of 96.9% and 94% respectively. While this project doesn't directly have a business application since it is a video game, following this approach using real-life statistics and metrics to build a model to predict player value or performance could be useful to sports clubs.

Some possible next steps would be to look at other positions other than attacker and use a similar approach to see if it would translate to other positions. Also, potentially combining other data sets from other years would potentially help us build a stronger model with a larger dataset. Lastly, gathering real-life statistics from players and trying a similar approach beyond video game ratings to see how well it would translate to actual players.