Fantasy Premier League Points Predictor

Data Visualization and Exploration

Contents

This presentation summarizes the exploratory data analysis done for the data. We discuss the following:

1. Data Preprocessing
2. Distribution of variables
3. Relationship between dependent and independent variables
4. Feature Importances

Project Objective

Our objective is to predict the fantasy points each player will bring in next week based on their average statistics from the prior weeks. We aim to do so by preprocessing the data, removing multicollinearity, selecting features, and choosing an appropriate regression model.

Data

- The past 6 years and partial data for the ongoing premier league season.
- For each year, there are approximately 600-700 players whose data are monitored.
- For each player, there are nearly 15 numerical and categorical features that are tracked.

ML Techniques To Be Implemented

- Feature Selection: decision tree feature importance, lasso regression, and correlation plots
- Regression: decision tree regressors (random forests, boosted trees, etc.), linear regression models, and potentially neural networks

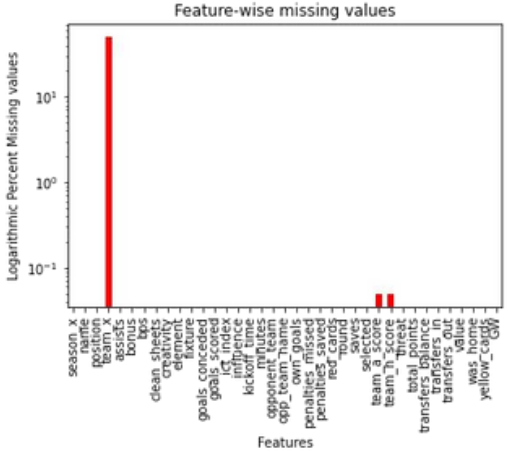
Data Preprocessing and Missing Values

Each row in the dataset represents the fantasy statistics of a player in a gameweek across the last 6 premier league seasons (one season is roughly a year long). The information can be categorized as:

1. Team-level statistics/information (opponent team, home game indicator, goals conceded)
2. Player-level statistics/information (position, goals scored, assists, saves, bonus points, etc.)
3. Game information (kickoff time, season, gameweek, home team score, away team score)

Predicting the next week's points based on the current week's statistics seems a little unfair to the player as it is possible that the player might not have performed well due to some reason. The reason could be a tough opponent, a slight niggle, or simply bad luck. To eliminate this, we take the average of numerical attributes from last three games (highlighted in red in the code), and then shift the data such that the average data corresponds to the points scored by the player in the current week.

```
# preprocessing
df = pd.read_csv('data/cleaned_merged_seasons.csv', index_col=0)
df = df.sort_values(['name', 'season_x', 'GW']).reset_index(drop = True)
X = df.loc[:, df.columns != 'total_points']
y = df.loc[:, ['name', 'total_points']]

X[avg_columns] = X[avg_columns].groupby('name', as_index=False).rolling(3, min_periods =
1).mean().reset_index(drop = True)
features = X.set_index(['name'], drop = True).groupby('name',
as_index=True).shift().dropna().reset_index()
y = y.reset_index(drop = True)
target = y.groupby('name').apply(lambda group: group.iloc[1:, :]).reset_index(drop =
True).drop(columns = ['name'])

df = pd.concat([features, target], axis = 1)

# missing values - fixture and kickoff time are also removed because they are irrelevant
df = df.drop(columns=['team_x', 'fixture', 'kickoff_time'])
df = df.fillna(0)
```



We also view the feature-wise missing values in the data using the plot on the right. We observe that team_x, team_h_score, and team_a_score have non-zero number of missing values.

1. Possibly, the players' team name was not stored prior to 2020-21 season. Since, a major chunk of this variable is missing values, and it doesn't primarily affect the objective performance of an individual player, we go on to drop the column from the data.

2. There are missing values in home team score and away team score for some players during GW 29 of 2019-20 season. We impute these missing values by zero as the situation is as good as each team scoring zero goals.

Multicollinearity and Correlations

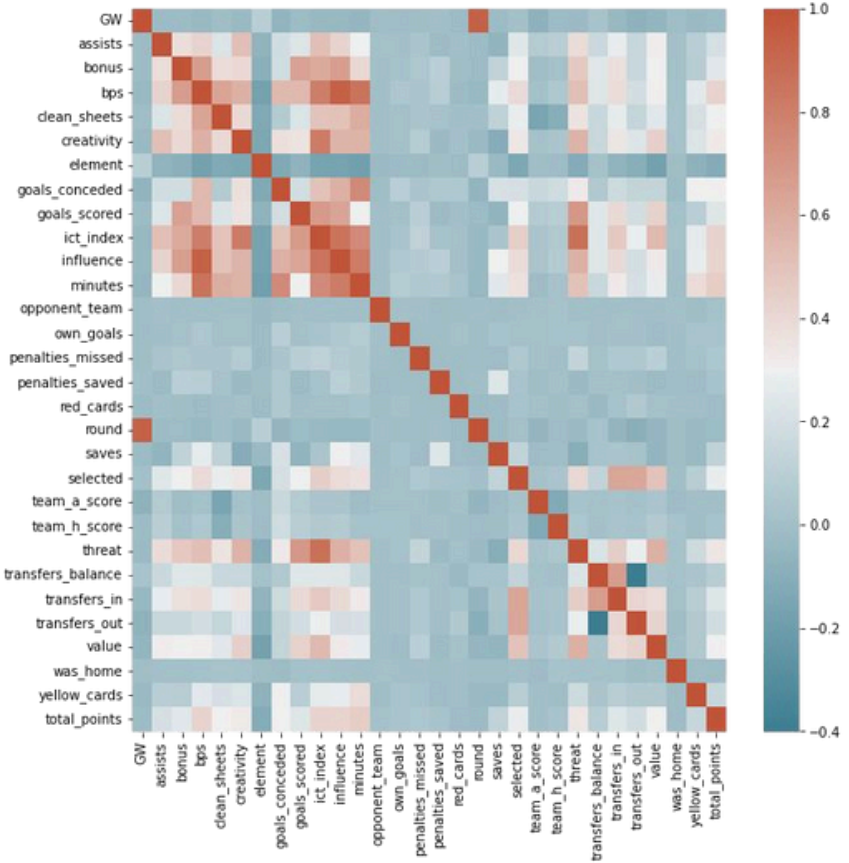
Given that we have numerous features and that our objective is to perform regression analysis, we find out highly correlated variables and drop features so as to remove multicollinearity. We observe that the following pairs of features are highly positively correlated:

1. Gameweek and Round: Redundant information.
2. Bonus Points (bps) and Influence: If a player is more influential, then the
3. chances of them getting bonus points are higher Bonus Points (bps) and Minutes: Usually, players who play a major strata of the game (in terms of minutes) are more likely to create a better impact (implying more bonus points). ict_index is simply a cumulative metric to measure influence, creativity, and threat, and is thus highly correlated with
4. influence and threat.

Highly Correlated Pairs

```
GW          round        0.935763
bps         influence    0.934918
            minutes      0.856918
ict_index   influence    0.850082
            threat       0.864673
dtype: float64
```

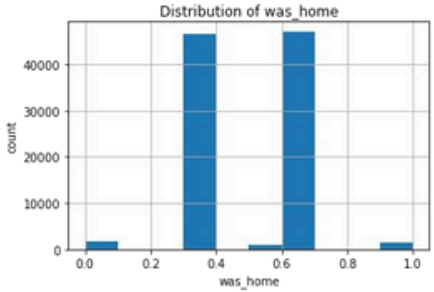
```
# heatmap of correlation matrix
df_corr = df.corr()
plt.figure(figsize=(10, 10))
sns.heatmap(df_corr, cmap="crest")

# Take a loot at the most highly correlated features
df_corr[df_corr >= 0.85].where(np.triu(np.ones(df_corr.shape), k =
1).astype(np.bool8)).stack()

# Remove multicollinearity

df = df.drop(columns=['round', 'influence', 'threat', 'minutes'])
```

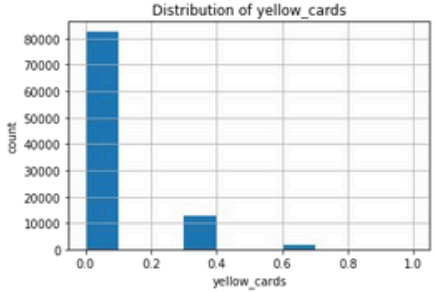


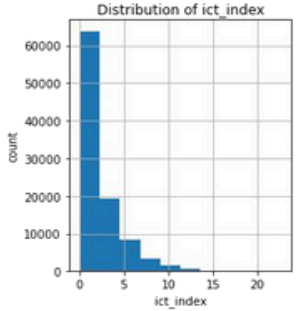
Data Distributions

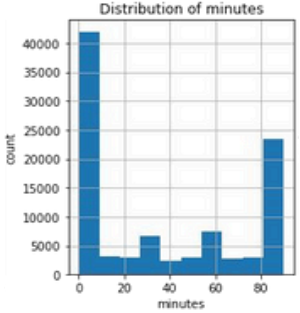
We plot distributions of all the variables (numerical, categorical, and target). Distribution of a few features are shown on the right. Note that, as discussed in the Data Preprocessing slide, we are considering a moving average of numerical statistics for our analysis. The plots are drawn for the following variables:

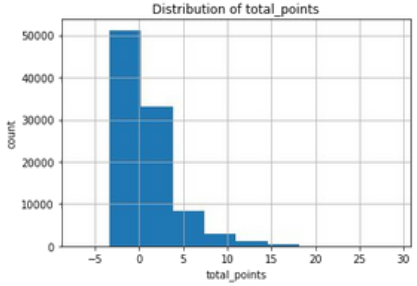
1. was_home: Since this is a categorical variable, we see discrete bars in the histogram plot.
2. yellow_cards: This is again a discrete variable and it is evident from the discrete bars in the histogram
3. ict_index: This variable is continuous in nature and we see that it is positively skewed.
4. minutes: The whole spectrum from 0-90 minutes is being covered as players are subbed in dynamically and clock different game time.
5. total_points: The target variable is also positively skewed, along with a few negative values.












```
# histograms
hist = df.hist(column=df.columns, layout=(5, 7), figsize=(30, 40))
for ax, feature in zip(hist.flatten(), sorted(df.columns)):
        ax.set_xlabel(feature) ax.set_ylabel('count') ax.set_title(f'Distribution
        of {feature}')
```

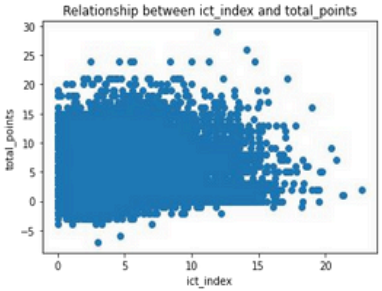
Relationship with Target Variables

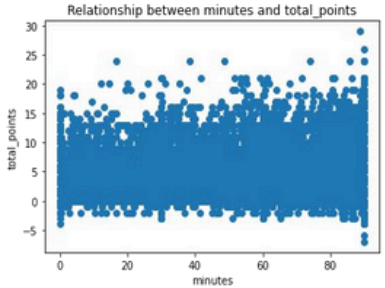
We plot feature relationships with the target variable using scatter plots for continuous variables and box plots for categorical features. A few plots have been shared on the right:

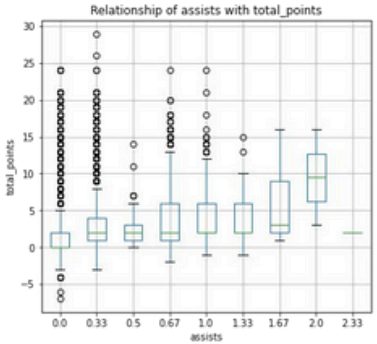
1. ict_index vs total_points: There seems to be a positive correlation between the feature and the target variable
2. minutes vs total_points: The scatter plot seems uniformly distributed. There seems to be no correlation between the number of minutes and the total points earned by the player.
3. assists vs total_points: We observe a number of outliers for people with zero assists but in general there is a positive correlation between the number of assists and the number of total points as visible from the rising median.
4. clean_sheets vs total_points: We again observe a number of outliers for 0 clean sheets as offensive players don't get any points for a clean sheet yet score high points when they score a goal. But in general, a positive correlation is observed between the two.

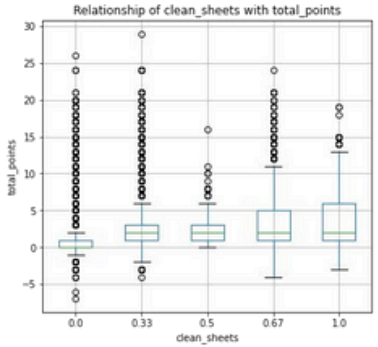
```
# scatter plots
for ax, feature in zip(axes.flatten(), sorted(numerical_vars)):
        ax.scatter(df[feature], df['total_points'])
        ax.set_xlabel(feature)
        ax.set_ylabel('total_points')
        ax.set_title(f'Relationship between {feature} and total_points')
```

```
# boxplots
for ax, feature in zip(axes.flatten(), sorted(categorical_vars)):
        df.boxplot(column=['total_points'], by=[feature], ax=ax)
        ax.set_xlabel(feature)
        ax.set_ylabel('total_points')
        ax.set_title(f'Relationship of {feature} with total_points')
```









Data Splitting (Sampling)

Our data comprises of all players' fantasy statistics arranged in a chronological order. Ideally, we would want our model to see some data for each player, preserve the time series behaviour, and then predict the number of total points based on the input features. Since we are averaging the numerical statistics using a moving window of size 3, the time series component is being captured and each entry can consequently be considered as independent.

Keeping in mind our use case, the ultimate goal is to design a team by choosing players who will maximize the points and adhere to budget constraints.

Therefore, for predicting the points of each player based on their most recent statistics, we would want the last rows of each players in the production test set. We will use structured splitting to do this. Although, we need not do this now. This can be done when we have finalized our model, and want to make predictions on unseen data which don't have labels.

For the purpose of building the model, tuning it, and evaluating the performance, we can do a simple random train-test split using scikit-learn's method. We propose a 60-20-20 split for training, validation, and test sets. Since, our objective is to predict the number of points a player can score solely on their past fantasy statistics, we can remove columns like name of the player, the opponent team, the gameweek and season along with previously scraped columns (team, fixture, kickoff-time) as they don't contribute to the prediction.

```
# dropping irrelevant columns
drop_cols = ['name', 'season_x', 'opponent_team', 'opp_team_name', 'GW']
df = df.drop(columns=drop_cols, errors='ignore')

# Data splitting 60-20-20
X_dev, X_test, y_dev, y_test = train_test_split(df.loc[:, df.columns != 'total_points'],
                                                df.loc[:, 'total_points'],
                                                random_state=42,
                                                test_size = 0.2)

X_train, X_val, y_train, y_val = train_test_split(X_dev,
y_dev,
random_state=42,
test_size = 0.25)
```

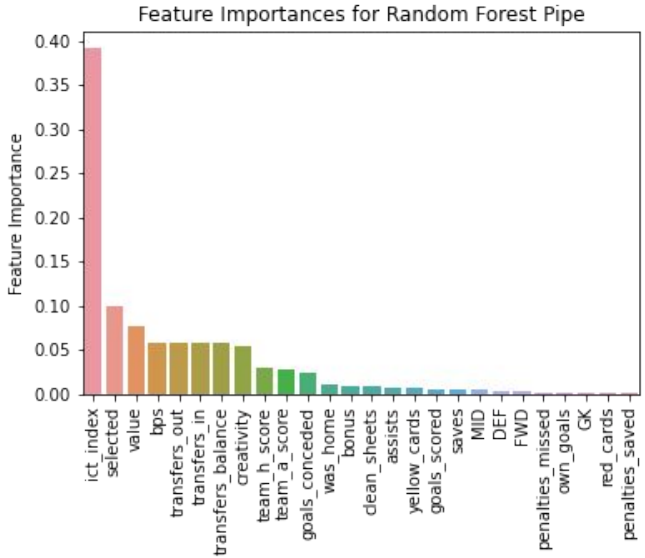
Columns remaining after all preprocessing:

['position', 'assists', 'bonus', 'bps', 'clean_sheets', 'creativity', 'goals_conceded', 'goals_scored', 'ict_index', 'own_goals', 'penalties_missed', 'penalties_saved', 'red_cards', 'saves', 'selected', 'team_a_score', 'team_h_score', 'transfers_balance', 'transfers_in', 'transfers_out', 'value', 'was_home', 'yellow_cards', 'total_points']

Note that only 'position' is the only categorical variable.

Feature Importance using Random Forest

We initially handle the categorical variable by one-hot encoding the position variable. We further fit a random forest regressor without any tuning, and then extract the feature importance calculated as the average drop in purity by a node. We observe that features like ict_index, selected, value, bps, creativity are amongst the most important features. These features being the most important make sense as they are a direct indicator of a player's performance as well as fantasy gamers' sentiment towards the player. This is just a preliminary feature importance analysis. In the next section, we provide feature selection using lasso regression.




```
# fit model
preprocess = make_column_transformer((OneHotEncoder(handle_unknown="ignore"),
['position']), remainder="passthrough")
pipe = make_pipeline(preprocess,RandomForestRegressor(max_depth=15))
pipe.fit(X_dev, y_dev)
```

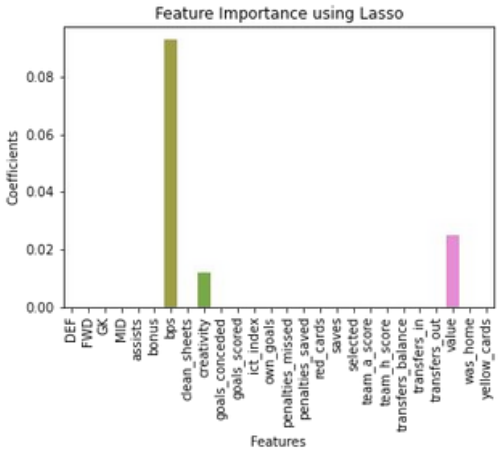
```
# Build the list of feature names
ohe_feature_names = preprocess.named_transformers_["onehotencoder"].get_feature_names().tolist()
ohe_feature_names = [x[3:] for x in ohe_feature_names]
feature_names = ohe_feature_names + list(df.loc[:, df.columns!='total_points'].drop(columns=['position']).columns)
# Get the importance of features
feat_imps = zip(feature_names, pipe[1].feature_importances_)
feats, imps = zip(*(sorted(list(filter(lambda x: x[1] != 0, feat_imps)), key=lambda x: x[1], reverse=True)))

# Plot the results
ax = sns.barplot(list(feats), list(imps))
ax.tick_params(axis='x', rotation=90)
ax.set_title('Feature Importances for Random Forest Pipe')
ax.set_ylabel('Feature Importance')
plt.show()
```

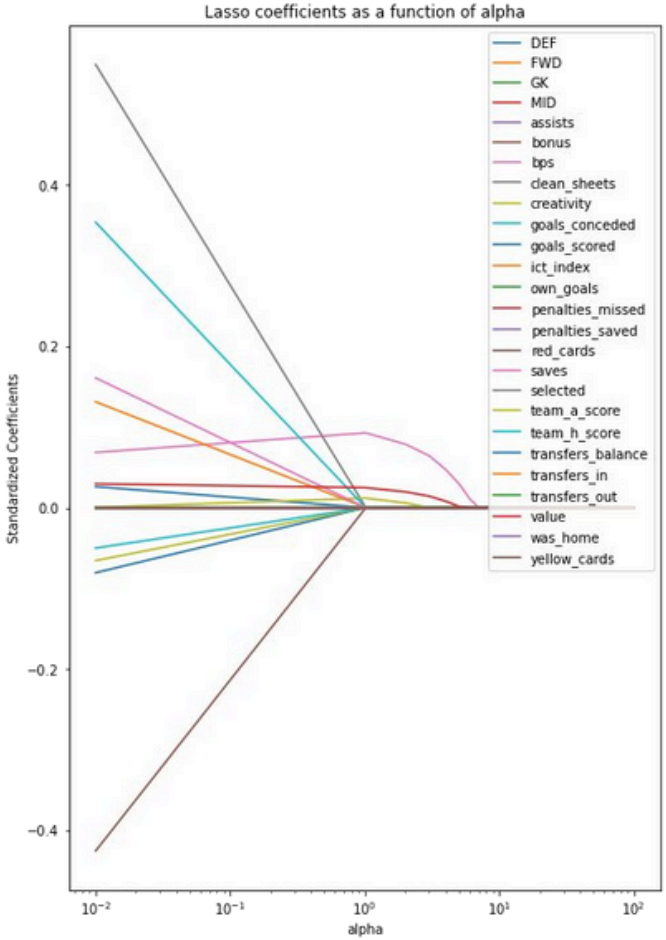
Feature Importance using Lasso Regression

The features set is large, and the distribution of feature importances shows that it contains features with importance closed to 0. We apply Linear Regression with l1 regularization (Lasso) to find the features whose weights reduced to 0 fast. The plot is shown on the right.

```
alphas = np.linspace(0.01,100,100)
lasso = Lasso(max_iter=1000)
for a in alphas:
        lasso.set_params(alpha=a)
        lasso.fit(preprocess.transform(df.loc[:,df.columns != 'total_points'],
                                                    df['total_points'])
```

We also plot feature importances on unprocessed data to find feature importances. Although, we remove some categorical variables like name, team names, etc. From the plot on the right, we can observe that there are tons of features whose coefficients are zero. The features with non-zero coefficients are bonus points, creativity, and value. These make sense in real-life as they are great determinants of a player's form and recent performance.



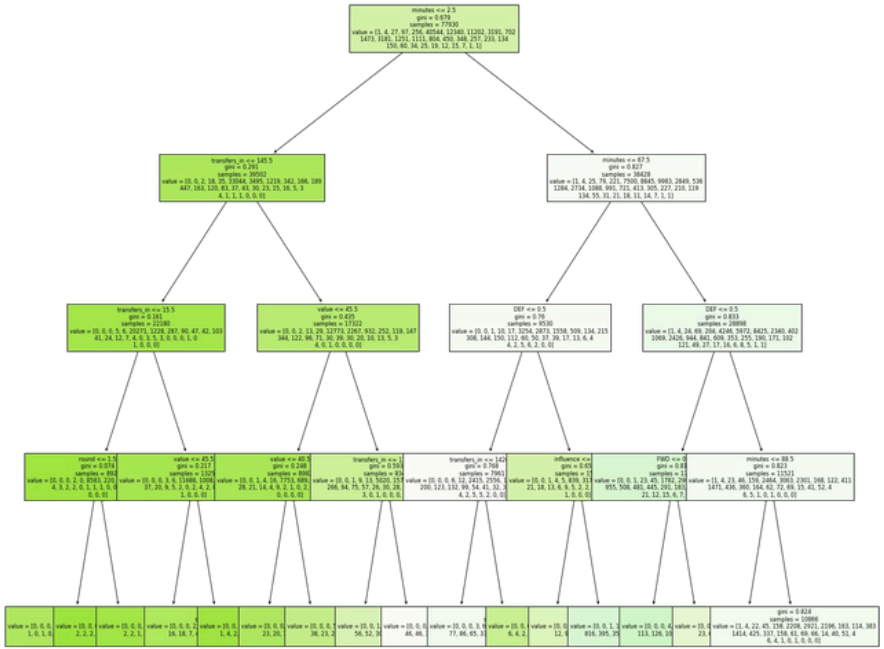


ML Techniques

Our starting point would be using decision tree regressors. For these regressors, we will be experimenting with different variations including random forests and boosted trees. Also, we will be experimenting with different hyperparameters for each model using grid search and random search. An example exploratory decision tree is shown on the right.

In addition, we will also be employing linear regression models with regularization and neural network models. From all the models explored, we will then compare models using various metrics such as mean absolute error (MAE), mean squared error (MSE), and mean absolute percentage error (MAPE).

```
from sklearn.tree import plot_tree

# Searching depths
param_grid = [{"max_depth": [2, 3, 4, 5, 6, 7, 8, 9, 10]}]

# Fit the random forest classifier
pipe = make_pipeline(preprocess, GridSearchCV(estimator=DecisionTreeClassifier(), param_grid=param_grid))
pipe.fit(x_dev, y_dev)

# Get the optimal parameter
grid_search_results = pipe.named_steps["gridsearchcv"]
print("Optimal depth: " + str(grid_search_results.best_params_['max_depth']))

# Rebuild the classifier using optimal depth
pipe = make_pipeline(preprocess, DecisionTreeClassifier(max_depth=grid_search_results.best_params_['max_depth']))
pipe.fit(x_dev, y_dev)

# Test accuracies on the most recent 20% of data
y_pred = pipe.predict(x_test)
print("Mean Absolute Error: " + str(mean_absolute_error(y_test, y_pred)))

# Plot the tree
plt.figure(figsize=(20,20))
plot_tree(pipe[1], filled=True, max_depth=8, fontsize=8, feature_names=feature_names)
plt.show()
```



Decision Tree Regressor

Thank you!

Applied Machine Learning