

Analyzing Scoring Efficiency of Players in the NBA with Machine Learning on Kubeflow

Introduction

A good look at the NBA opening playoffs between Denver Nuggets and the Utah Jazz, which took place in August 2020, will have you awestruck by the absolute masterclass of scoring ability and the willpower to win. But the spotlight was definitely on two players; Donovan Mitchell of Utah Jazz and Jamal Murray of Denver Nuggets. Both players played to record-breaking heights, setting an all-time high for points scored by opposing players in a playoff series. Donovan Mitchell bagged 51 points and Jamal Murray scored 50 points in the crazy duel in Game 4 of the NBA playoffs.

Like any other professional sports, Analytics in the NBA is revolutionizing the way we think and talk about basketball. The NBA has adopted analytics significantly by collecting data using cameras that record every movement of both the ball and all ten players 25 times per second. It has become more obvious to fans how the adoption of analytics in basketball has brought about an explosion in attempts for three-point shots.

For instance, teams averaged about 18 three-point attempts per game in 2012. That number had increased to 42.8 in 2020. The reason being that insight from data showed that the reward of taking a three-point shot outweighed the risk. On average, teams that take more three-point shots ultimately score more points over the course of the game.

Apart from the changes on court, the locker room and training grounds have received various improvements on the back of modern utilization of data analytics

concepts. Play tactics, medical data on fatigue and injuries help players perform better, with longer careers and more time playing at their peak.

In this project, we took a look at a season of NBA shot data, exploring player efficiency and shot selection.

Data

The dataset we used is from [Kaggle](#) and it contains shots taken during the 2014-2015 season from October 2014 to March 2015, with data on 281 players and 473 defenders about who took the shot, where on the floor was the shot taken, who was the nearest defender, how far away was the nearest defender, time on the shot clock as well as some other interesting game data.

Analysis

Looking at the NBA games as a whole, there has always been an element of seasonality in the performances of the players, with both managers and the fans expecting better in the playoffs and around certain events. Our first analysis explores this theme

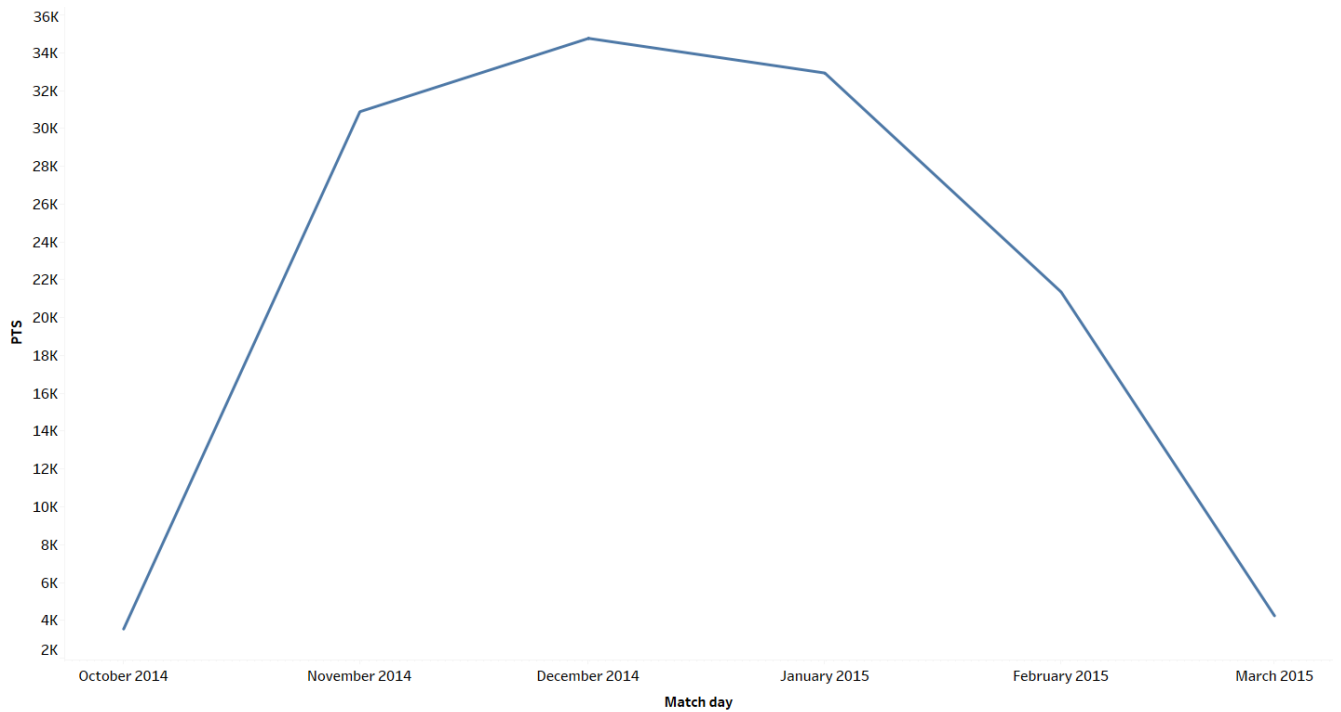


Figure 1: Number of points scored from October 2014 to March 2015

In figure 1 above, the number of high scoring games reached a peak around December with over 34 thousand points coming from teams playing during that period. We can observe a ramp-up process from the start of the season that drops off in the last three months. The assumption is that apart from the difference in fixtures during those months, as team standings get more stable, managers and players look to avoid injuries in the days leading to the play-offs.

In terms of shooting, two-pointers are often the go-to option for most players and teams. This is an approach that is slowly changing as the modern game acclimatizes to new players, from point guards to centres, who can all take a shot from outside the arc. Here we explored the number of points, whether two or three-pointers and the shot clock at the time of scoring.

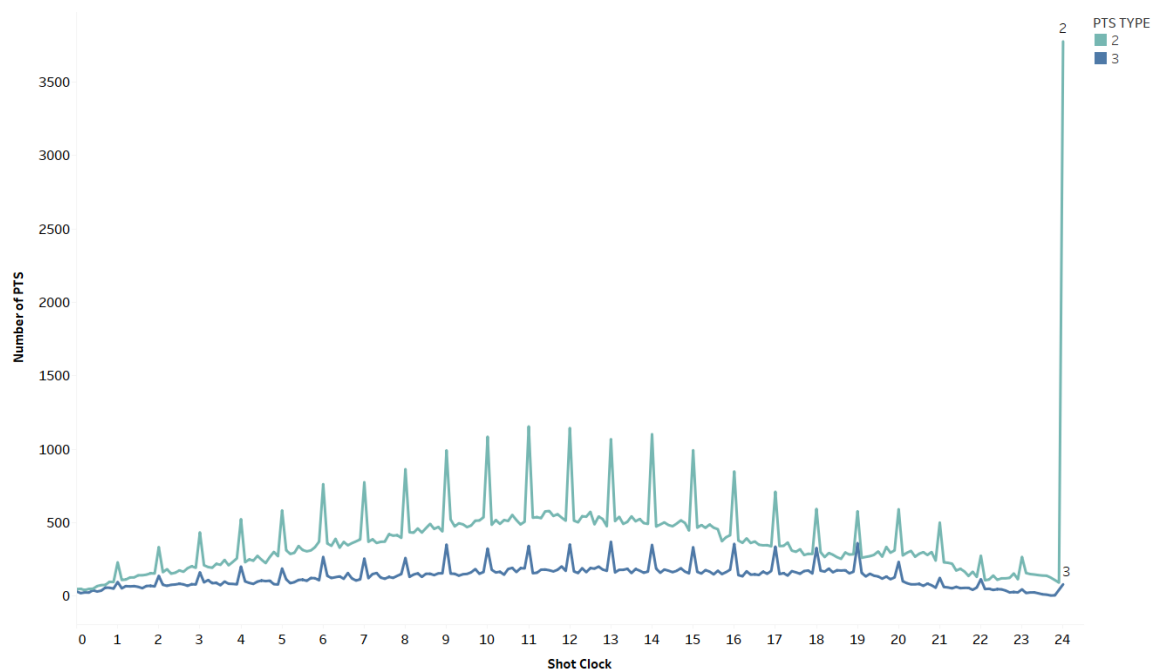


Figure 2: Number of 2-pointers and 3-pointers recorded at the different shot clock intervals

From figure 2 above, it is obvious that the two-pointer is obviously dominant at every time period, as most plays will prioritise an easier shot in the painted area. The large spike at the 24 seconds period can be attributed to offensive rebounds with players putting the ball in the basket immediately after a missed shot restarts the clock.

Speaking of quick points, NBA players often train themselves to shoot quickly, in a free-flowing game with the ball moving around as players switch positions, an uncontested (“open look”) chance at the basket is always fleeting with the nearest defender a few seconds away. Alternatively, the assumption will be that the more time a player has to gauge their shot, the better their chances of getting a favourable result, we looked at the average touch time and the results of the shot in the next visualization.

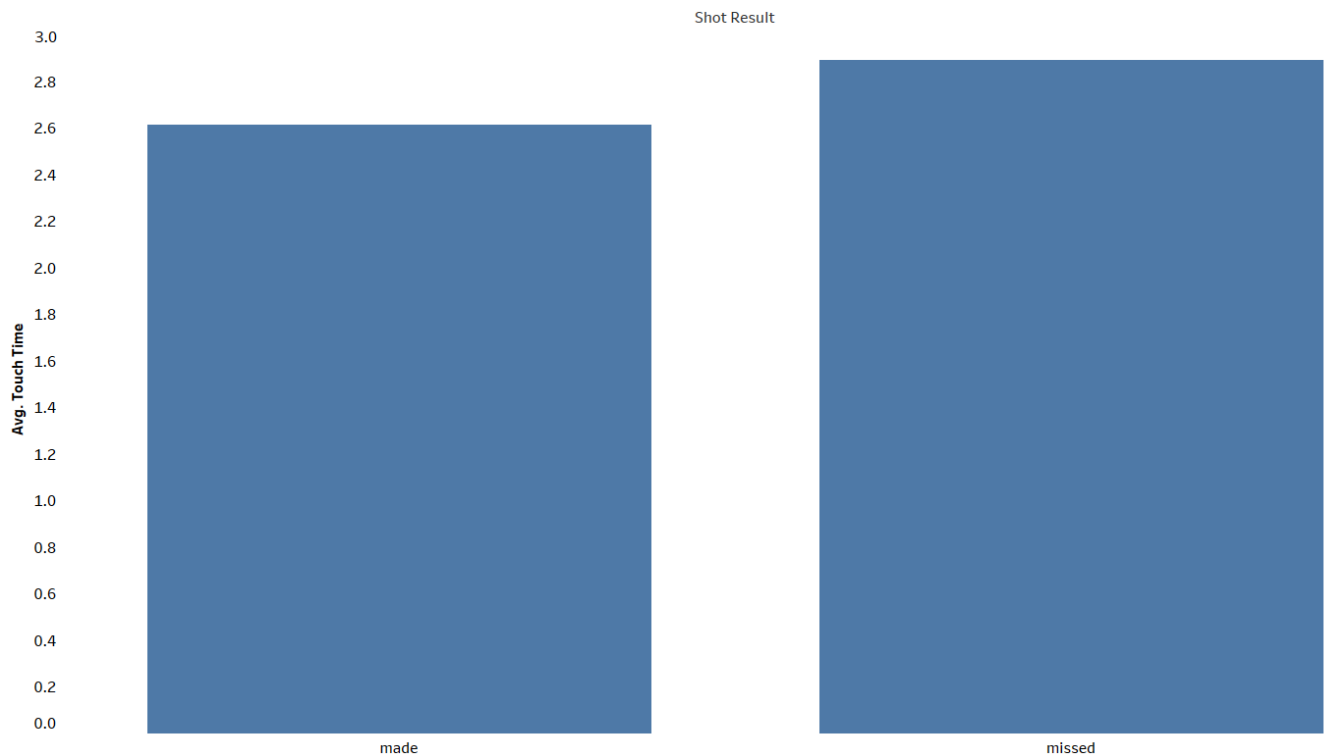


Figure 3: Bar chart showing the average touch time vs shot result

From our visualisation shown in figure 3, we can see that on average more time with the ball did not lead to better shot chances. This could be because of the catch and shoot mentality of players in open looks and an indication of the quality of defenders in the NBA with close defenders often forcing bad shot selections.

Attributing the chances of missing to the nearest defender, we can look at who the 'best' defenders are, by exploring the number of misses from all players in the

season.

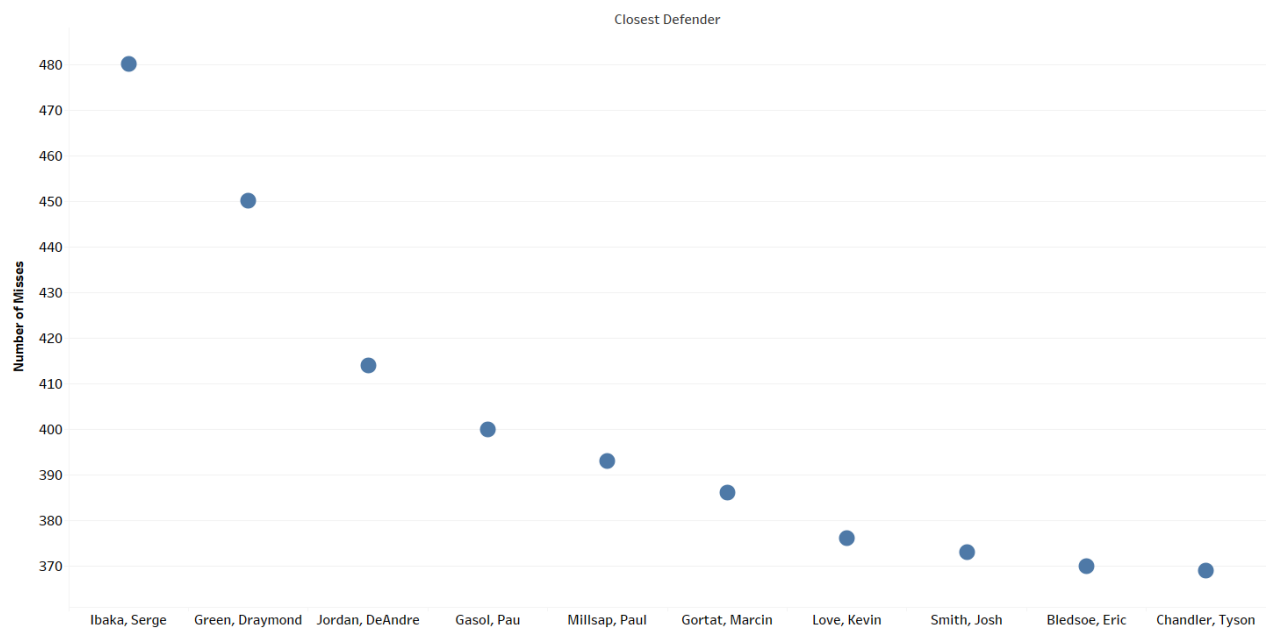


Figure 4: Number of misses vs closest defenders responsible for the misses

Figure 4 shows that Serge Ibaka leads the pack with 480 misses followed by Draymond Green and DeAndre Jordan. These three, especially Ibaka, were in a class of their own being the lead defenders for a significant amount of misses in the season.

Returning to our analysis of the offensive end of the NBA season, we can take a look at the top ten players in terms of points scored during the season.

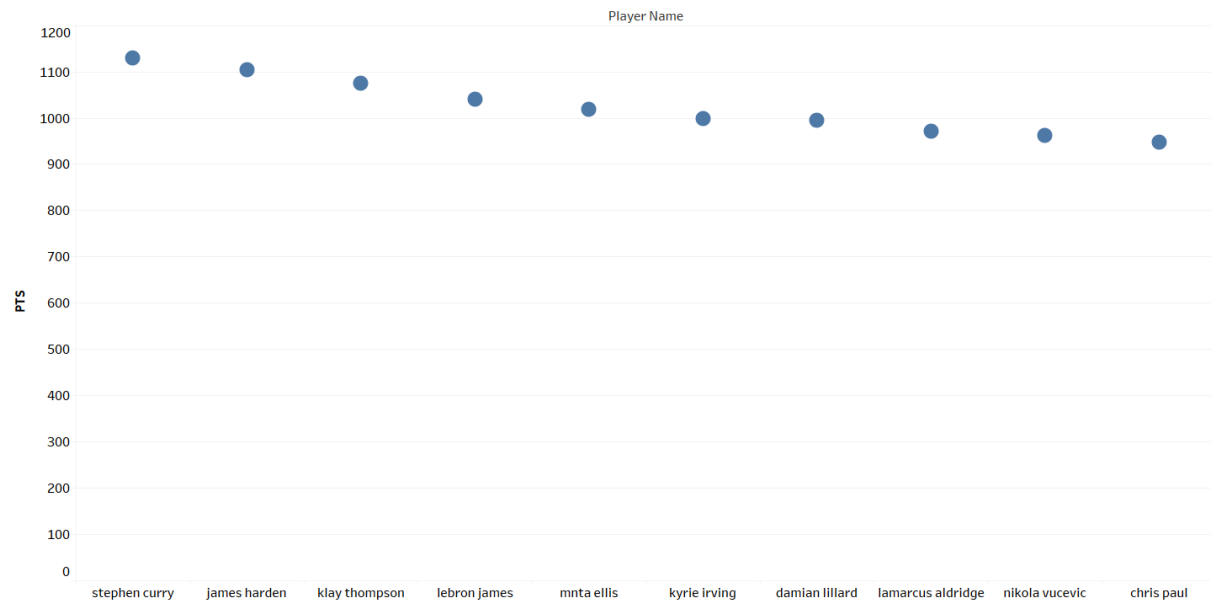


Figure 5: Top ten scorers in the NBA season

This visualization in figure 5 shows the NBA season's most prolific scorers with Stephen Curry showing up first. Steph is well known for his scoring and the 2014-2015 season started off a series of high flying performances for him, as he won his first Most Valuable Player (MVP) award and led his team Golden State Warrior to their first playoff championship in 40 years. The second-generation NBA superstar went on to win his second MVP and two more playoff titles in the three subsequent years.

Over the next visualizations, we will explore what differentiates Stephen Curry from other scorers. First, we examine the number of shots and the corresponding points made.

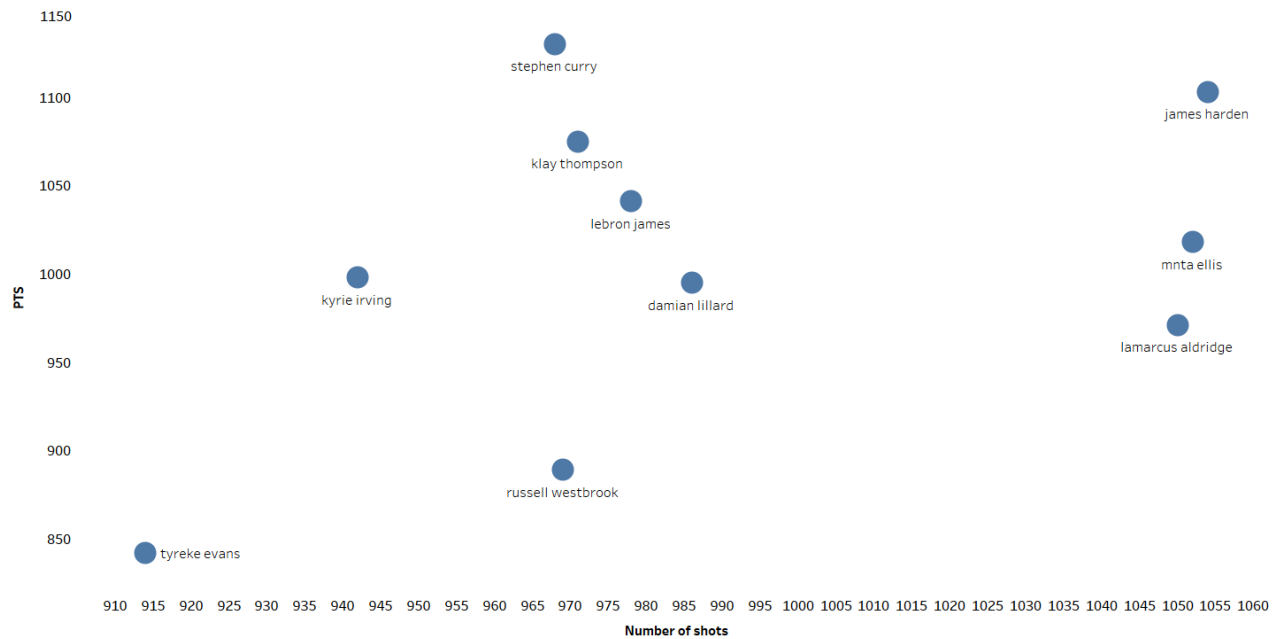


Figure 6: Number of points scored vs number of shots made

Almost all of our previously visualized top 10 scorers are captured in this visualization in figure 6 with Chris Paul swapping for Tyrese Evans based on the number of shots taken.

Most players came short of the 1000 shot mark with only three players attempting more shots in the season. Stephen Curry comes up in the bottom four on shots attempted.

This difference in shot attempted to points made could point to a better shot selection or more value for each shot made. Engaging in some comparative analysis, we explore the field goal percentages from Steph Curry and the rest of the NBA players.

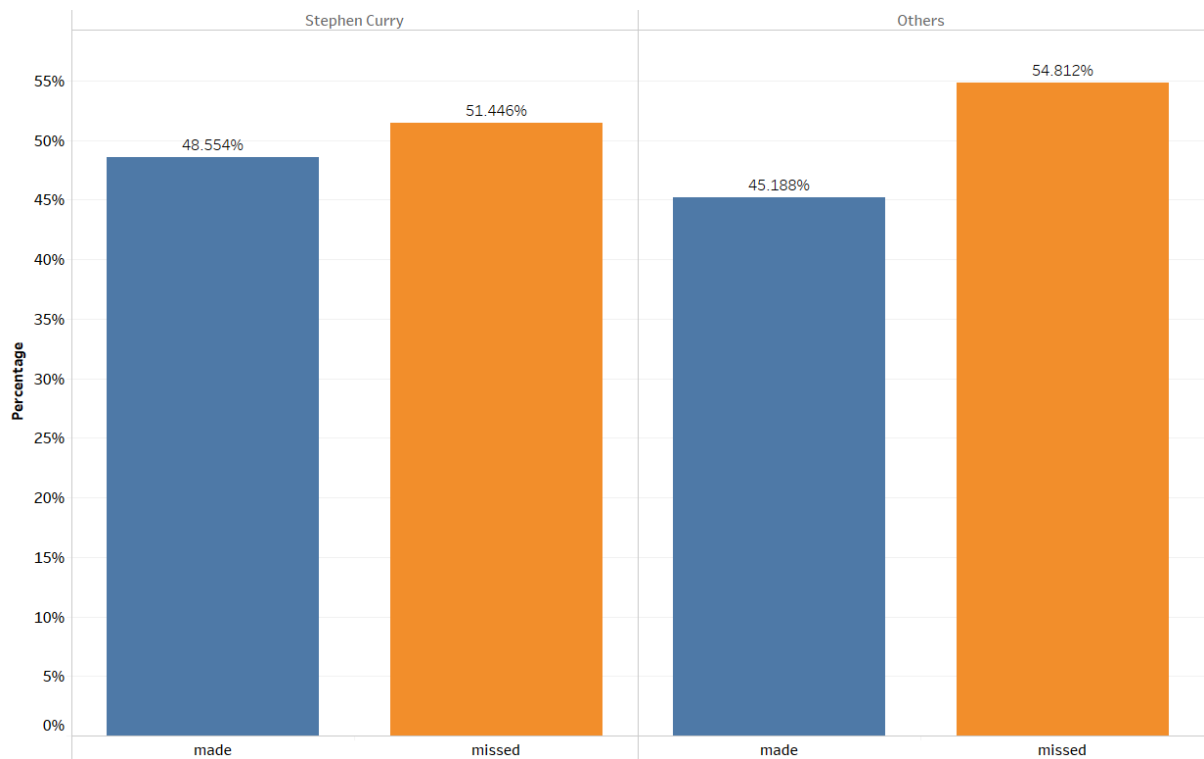


Figure 7: Percentage of shots made and missed by Stephen Curry compared to the other players

Making more shots than the average player as seen in figure 7 above, Stephen Curry had close to 50 percent of his attempted shots going through the basket.

Stephen Curry, over the course of his career, has been an exemplar in his three-point shooting, looking at the average shot distance throughout the season, we compare performances between him and the average player.

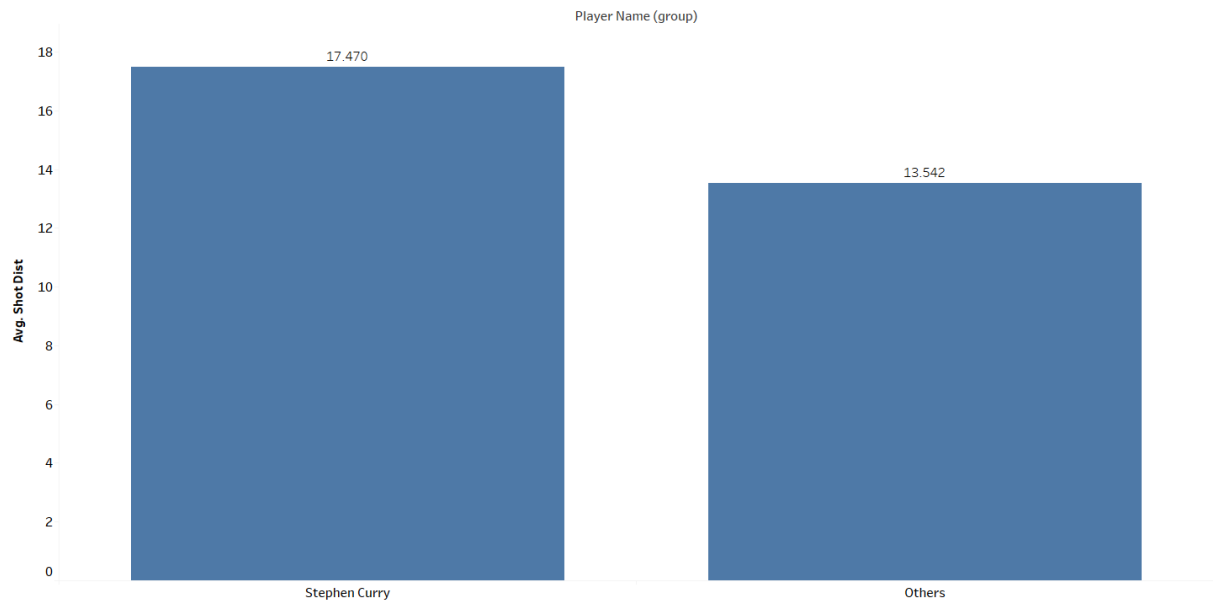


Figure 8: Average shot distance of Stephen Curry compared to other players

Playing as a point guard, Stephen Curry has had the opportunity to play above the arc and as a frequent three-point shooter, his average shot distance is well beyond the free throw line, when compared to his fellow players as seen in figure 8 above.

This accuracy even when shooting from farther away could be seen as a key part of his scoring dominance. This is because shots taken beyond the arc are valued higher, each shot increasing the impact on the game.

However, Curry is far from the only player with the three-point shooting ability.

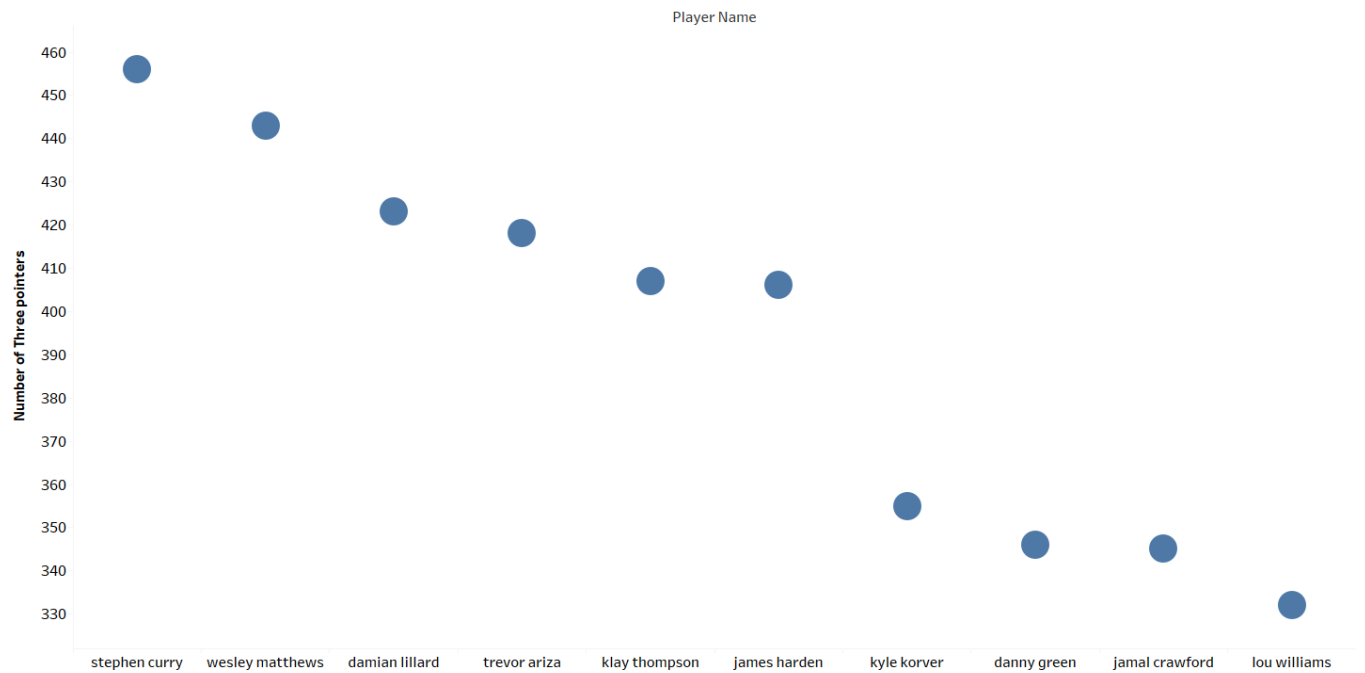


Figure 9: Top ten players with the most number of three-pointers made

Looking at the top 10 players based on the number of three-points made in the season from figure 9 above, Stephen Curry again leads the pack, however, only three other names; Damian Lillard, Klay Thompson and James Harden, show up on the top 10 for points scored. This calls into question accurate three point shooting as the sole reason for success.

Comparing the top three 3-point shooters with the averages from the rest of the NBA players, we examine differences in playing styles and performance.

Our first foray analyzes the average distance of the closest defender and the average number of dribbles before the shot is made.



Figure 10: Average closest defender distance vs average number of dribbles

Oftentimes a shooter dribbles before a shot to gain some distance from their defender, alternatively, players who rely on open or uncontested shots have less reason to dribble. Looking at the above visualization in figure 10, we can posit some extremes with Wesley Matthews having the highest average shot distance and the least average number of dribbles and Damian Lillard having the highest average number of dribbles and the lowest average distance from a defender of the three point shooters. Stephen Curry is near the middle of both playing style extremes, being closer to Lillard. We could also attribute this visualization to positional differences as both Lillard and Curry play mostly as Point Guards whose primary objective is to create scoring opportunities for the team while Matthews plays as a

Shooting Guard/Small Forward, both positions that are traditionally targets for scoring opportunities.

Lastly, we explore the average shot distance and the average time period a shot is taken.

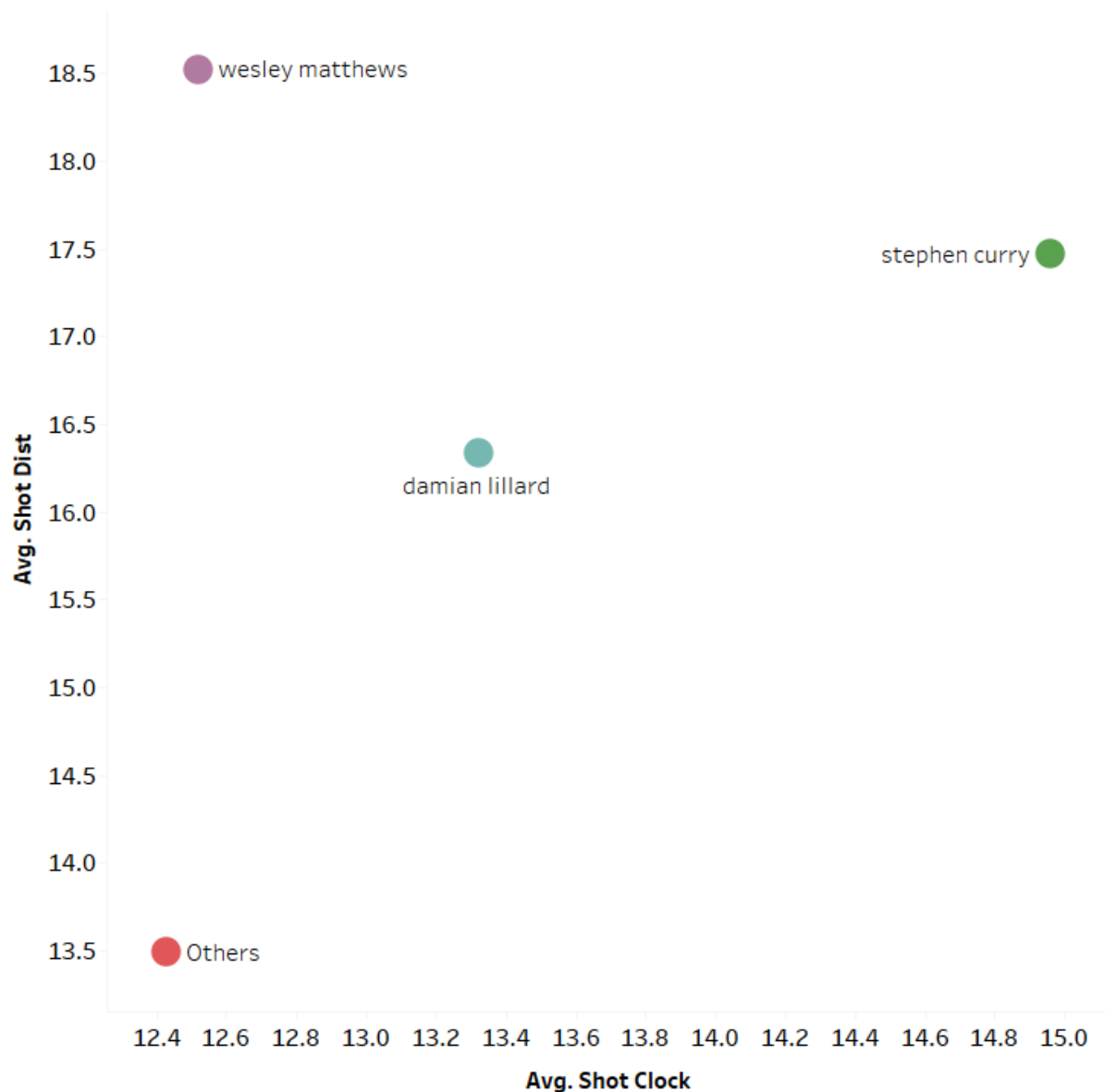


Figure 11: Average shot distance vs Average shot clock

The three-point shooters all shoot on average from above the free-throw line, Curry is seen to score earlier in the shot clock session taking only about 15 seconds of the available 24 to find a shot on average as shown in figure 11 above.

Data Processing

To make our data suitable for the machine learning models, we performed the following preprocessing steps:

1. Dropping rows with missing values
2. Converting categorical features to numerical values
3. Dropping extraneous columns and columns with highly [correlated](#) features
4. Splitting the data into train and test sets

Modelling

We trained the data with Logistic Regression, Random Forest and Gradient Boosting Models using shot result (made or missed) as our target.

Our best performing model was the Gradient Boost Model which produced an accuracy of 62.67%

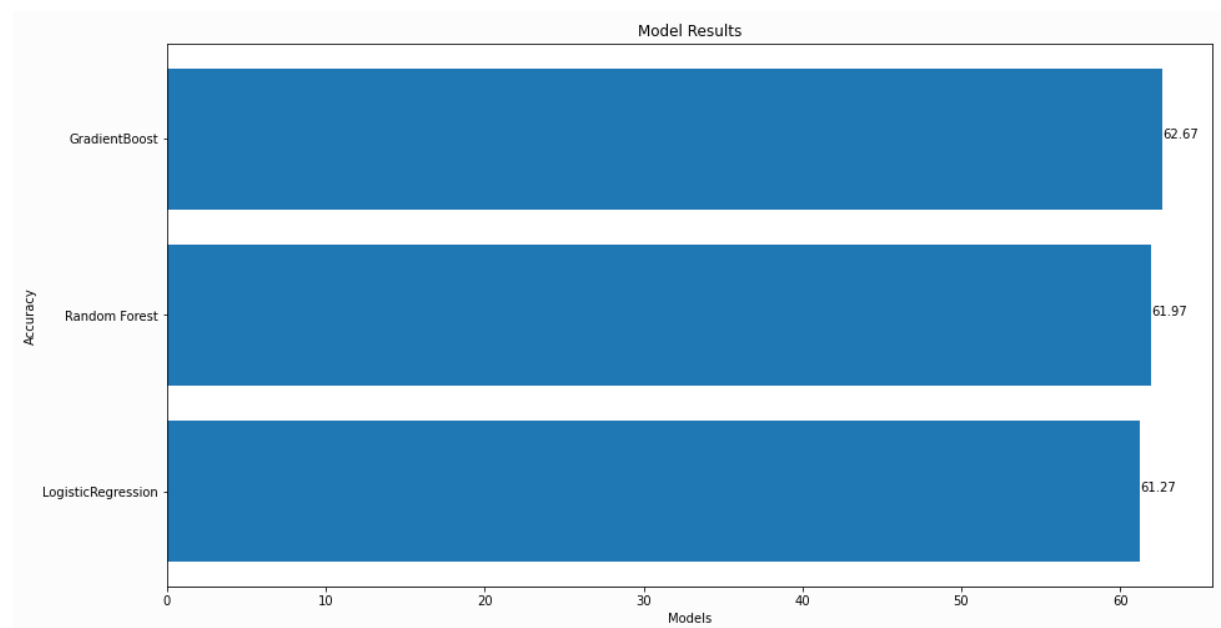


Figure 12: Model Accuracies

Model Explanation

Model Explainability has to do with the ability of the parameters to justify the results. It is the extent to which the internal mechanics of a machine or deep learning system

can be explained in human terms. In basic terms, Explainability is being able to quite literally explain what is happening.

Several tools are available for ME, we selected Alibi Explain for this purpose.

Alibi Explain

Alibi is an open-source Python library aimed at machine learning model inspection and interpretation. The focus of the library is to provide high-quality implementations of black-box, white-box, local and global explanation methods for classification and regression models.

In this project, we use the ALE (Accumulated Local Effects) explainer on the NBA dataset to illustrate Machine learning model explainability.

Accumulated Local Effects

One of the fascinating things about Machine Learning is that you can create a model by using different kinds of algorithms. Because of this freedom, however, models can be vastly different and complicated in unexpected ways. As such, *model agnostic* tools are needed to allow us to understand these models regardless of how they were created.

One of such tools is the [Accumulated Local Effects \(ALE\) Plots](#). It helps us determine the effect that each individual input, isolated from all others, has on our output.

So if we're creating a model to predict the efficiency of NBA players, and we include factors like field goals made, shot distance, point type, touch time, age of players, etc, an ALE plot would show us how much just the field goals made, for instance, affects the prediction, regardless of the other factors.

In this session, we will use an ALE explainer (Accumulated Local Effects) to explain the behaviour of the Regression models used in this analysis. on the NBA dataset.

Logistic regression

ALE on Logistic Regression in logit space

Below are the ALE plots for explaining the effects of each feature towards the unnormalized logit scores:

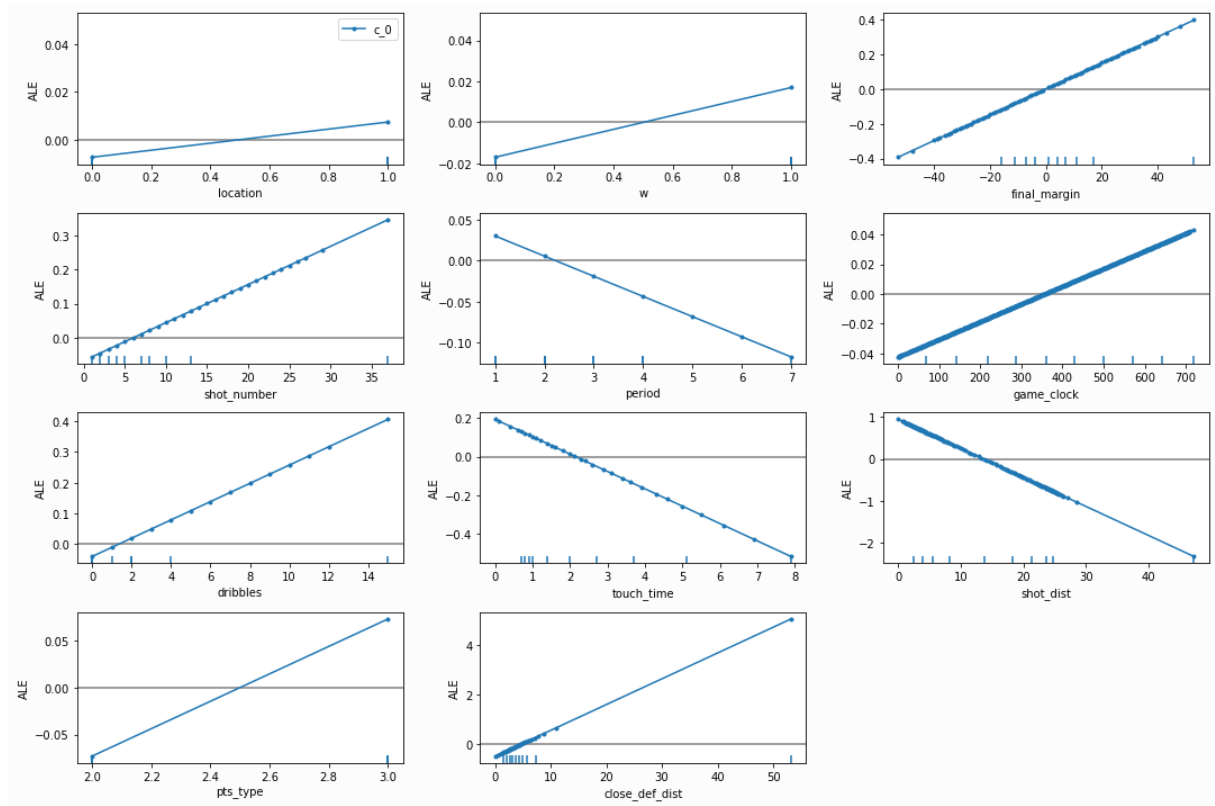
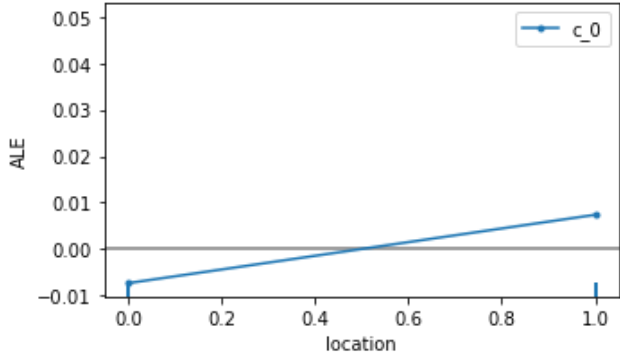
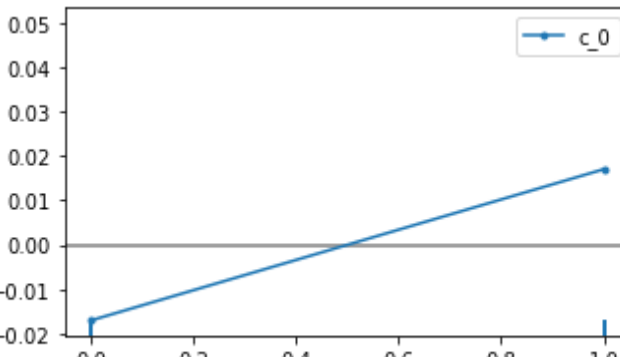
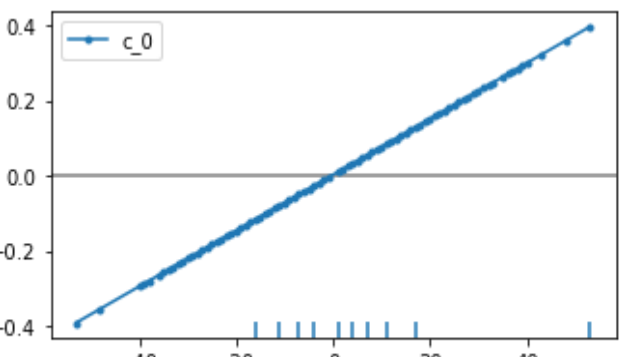
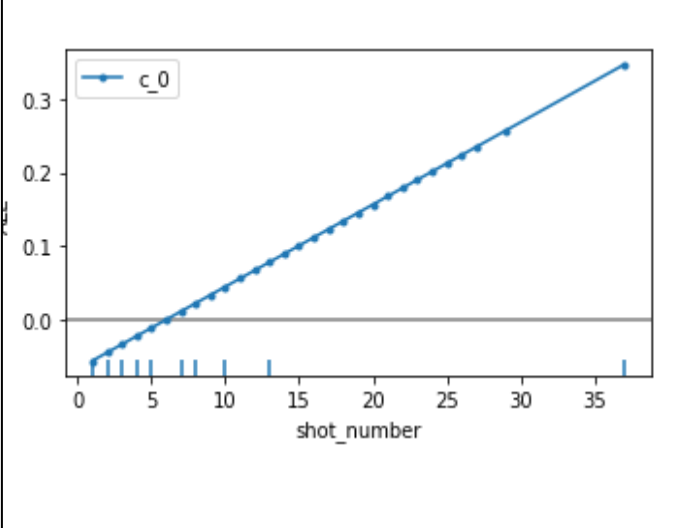
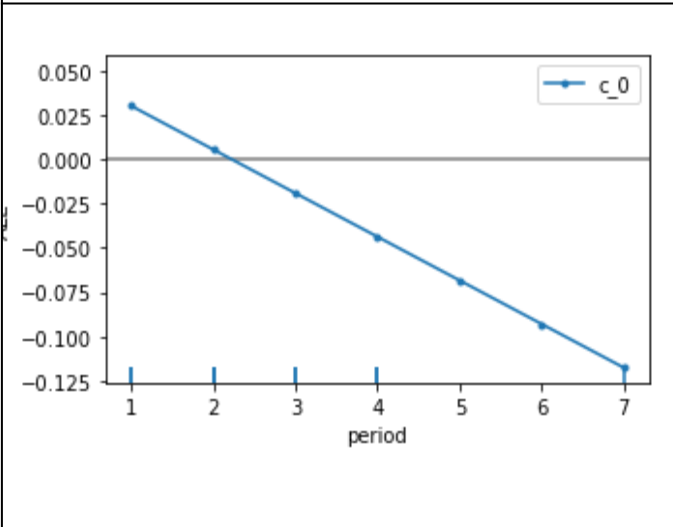
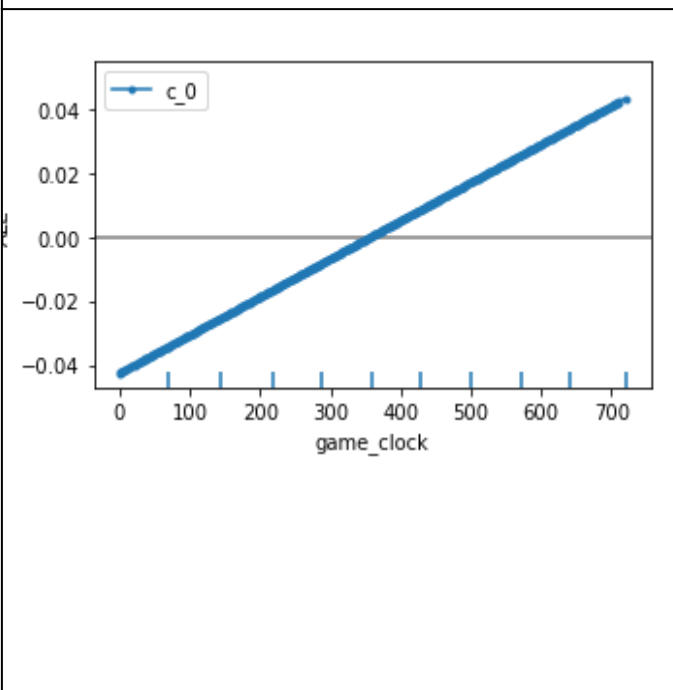


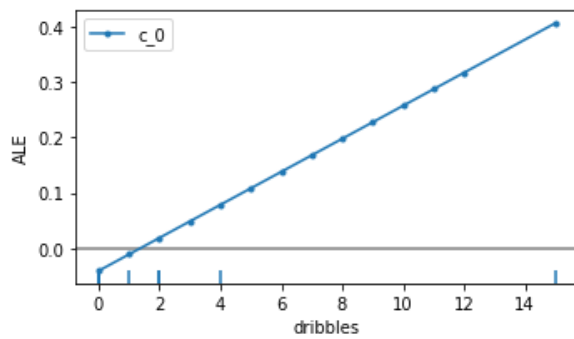
Figure 13: ALE on Logistic Regression in logit space

We see that the feature effects are linear for each class and each feature. This is exactly what we expect because the logistic regression is a linear model in the [logit space](#). Furthermore, the units of the ALE plots here are in logits, which means that the feature effect at some feature value will be a positive or negative contribution to the logit of each class with respect to the mean feature effect.

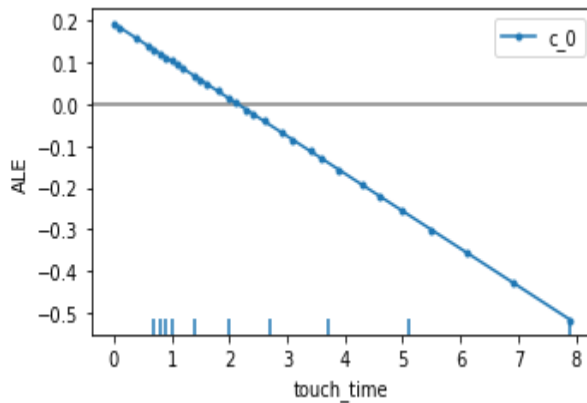
Let's look at the interpretation of each feature effect in more detail:

 <p>The plot shows the Average Leaning Effect (ALE) for the 'location' feature. The x-axis is labeled 'location' and ranges from 0.0 to 1.0. The y-axis is labeled 'ALE' and ranges from -0.01 to 0.05. A blue line with markers, labeled 'c_0' in the legend, starts at approximately (0.0, -0.008) and increases linearly to approximately (1.0, 0.008). A horizontal grey line is drawn at y=0.0.</p>	<p>Location - for locations 0(Away Games), the feature effect on the prediction is negative and for locations 1(Home Games), the feature effect on the prediction is positive</p>
 <p>The plot shows the Average Leaning Effect (ALE) for the 'w' feature. The x-axis is labeled 'w' and ranges from 0.0 to 1.0. The y-axis is labeled 'ALE' and ranges from -0.02 to 0.05. A blue line with markers, labeled 'c_0' in the legend, starts at approximately (0.0, -0.015) and increases linearly to approximately (1.0, 0.018). A horizontal grey line is drawn at y=0.0.</p>	<p>w - for games won (1), the feature has a positive effect on the average prediction, and for games lost(0), the feature has a negative effect on the average</p>
 <p>The plot shows the Average Leaning Effect (ALE) for the 'final_margin' feature. The x-axis is labeled 'final_margin' and ranges from -40 to 40. The y-axis is labeled 'ALE' and ranges from -0.4 to 0.4. A blue line with markers, labeled 'c_0' in the legend, shows a strong positive linear relationship, starting at approximately (-45, -0.38) and ending at approximately (50, 0.38). A horizontal grey line is drawn at y=0.0.</p>	<p>final margin - As the final margin increases, its effect on the average prediction also increases</p>

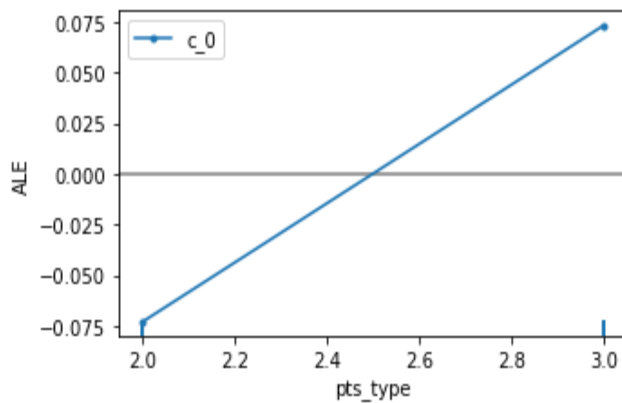
	<p>shot number - We see a positive correlation between the shot number and ALE values. An increase in shot number results leads to an increased effect on the average prediction</p>
	<p>period - longer periods have negative effects on the average prediction, while shorter periods have positive effects</p>
	<p>game clock - Game Clock is positively correlated with ALE values</p>



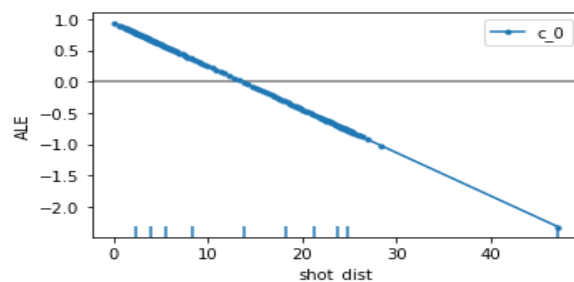
dribbles - At dribbles = 1 to 1.5, the effect on average prediction is negative. But its effect begins to increase from dribbles > 1.5



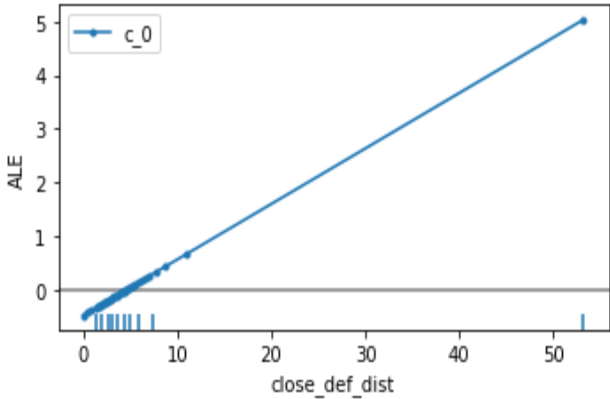
touch time - Smaller touch times have positive effects on the average prediction



points type - Point types of 2 affect the average prediction negatively, while point types of 3 have positive effects on average prediction



shot distance - The ALE lines cross the 0 mark at ~10.4 which means that for instances of shot distance around ~10.4, the feature effect on the prediction is the same

	<p>as the average feature effect. On the other hand, going towards the extreme values of the feature, the model assigns positive values for shot distances less than 10.4 and negative penalties towards classifying shots as missed for shot distances greater than 10.4</p>
	<p>close defender distance - the farther the closest defenders distance, the more effect it has on average prediction</p>

ALE on Logistic Regression in probability space

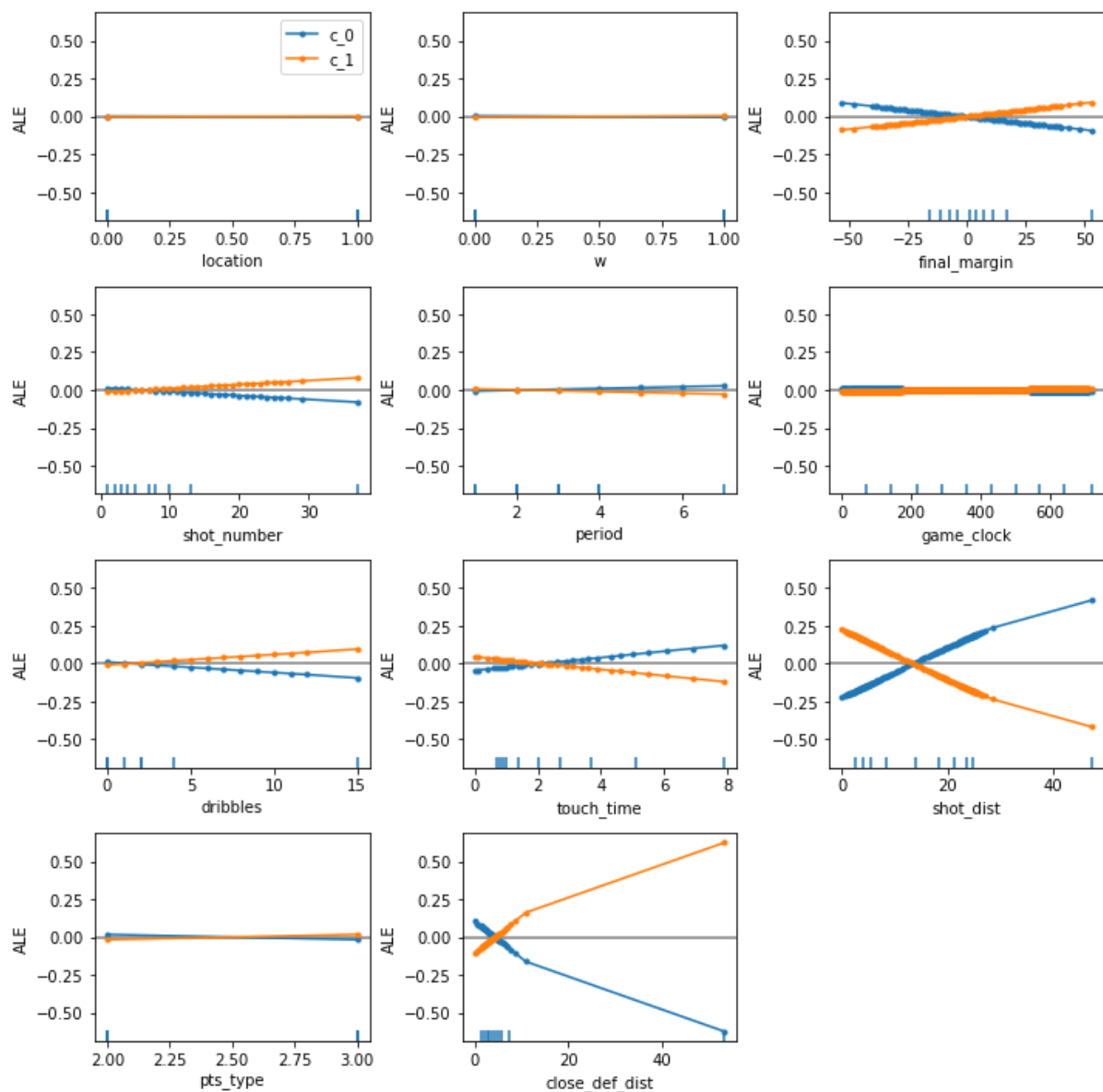
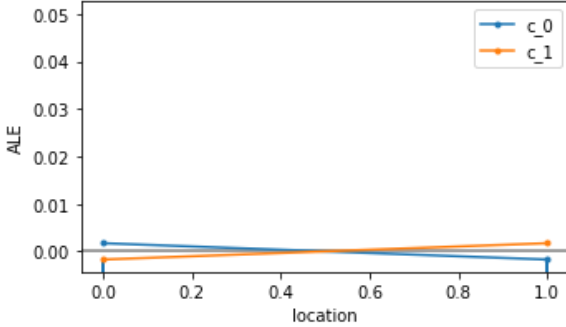
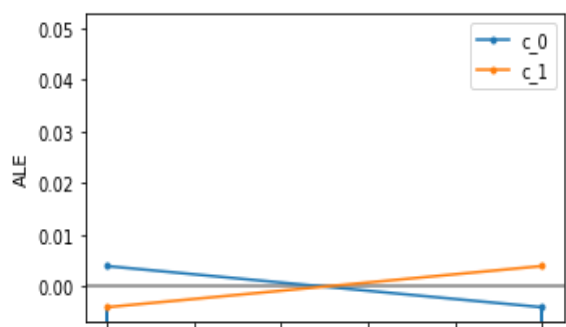
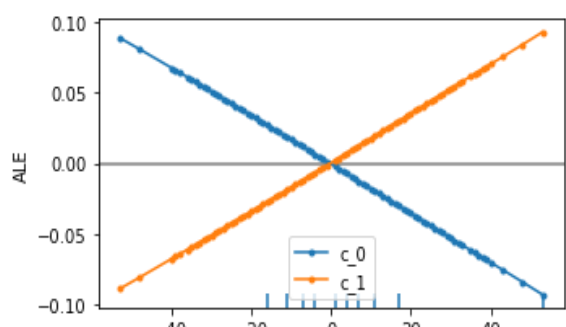


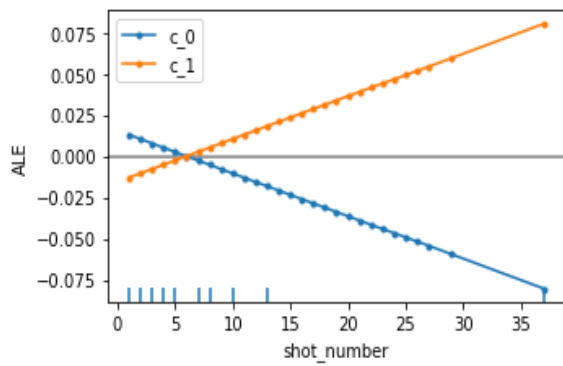
Figure 14: ALE on Logistic Regression in probability space

As expected, the ALE plots are no longer linear.

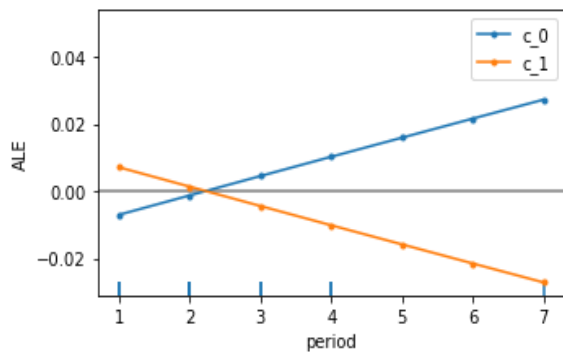
In this case, the ALE is in the units of relative probability mass, i.e. given a feature value how much more (less) probability does the model assign to each class relative to the mean prediction. This also means that any increase in the relative probability of one class must result in a decrease in the probability of another class. In fact, the ALE curves summed across classes result in 0 as a direct consequence of conservation of probability.

Let's take a closer look at the interpretation of each feature effect in more detail:

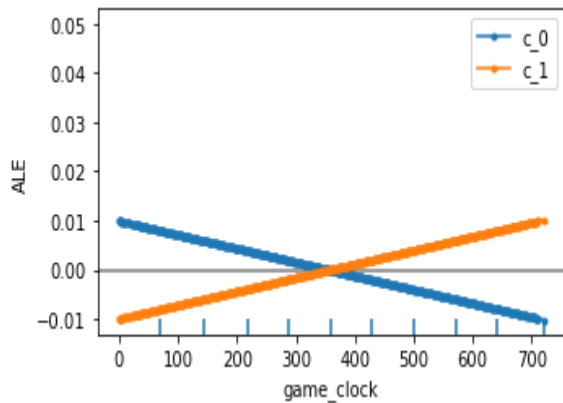
	<p>location - Location seems to have no effect on the prediction</p>
	<p>w - for won games(1), the model assigns positive probabilities towards classifying the instance as 1 and negative probabilities towards classifying the instance as 0. For lost games(0), the model assigns positive probabilities towards classifying the instance as 0 and negative probabilities towards classifying the instance as 1</p>
	<p>final margin - for final margins greater than 0, the model assigns positive probabilities towards classifying the instance as 1 and for final margins less than 0, the model assigns positive probabilities towards classifying the instance as 0</p>



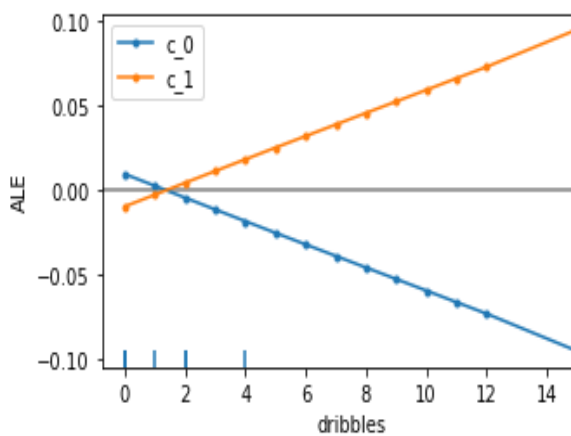
shot number - As shot number increases, the model assigns higher probabilities towards classifying the instance as 1



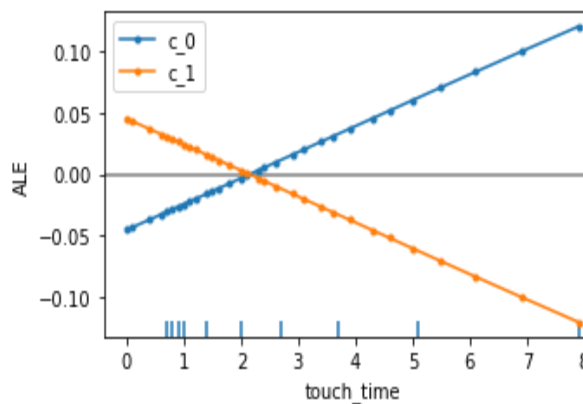
period - As period increases, the model assigns lower probabilities towards classifying the instance as a 1 and higher probabilities towards classifying it as a 0



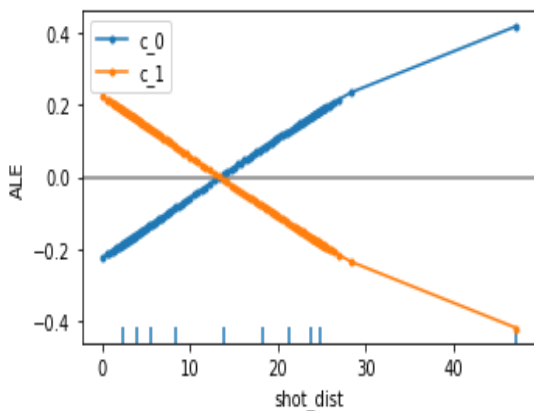
game clock -game clocks below 350 are being assigned negative probabilities towards being classified as 1 and game clocks above 350 are being assigned positive probabilities towards being classified as 1



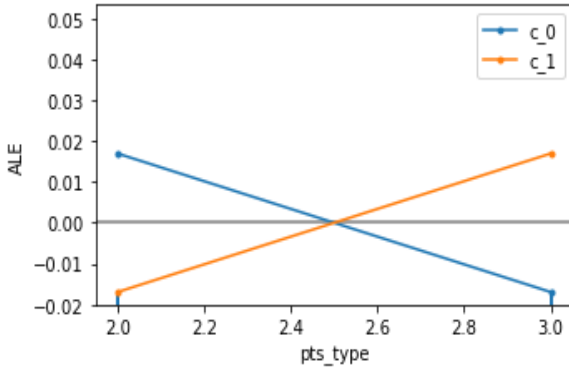
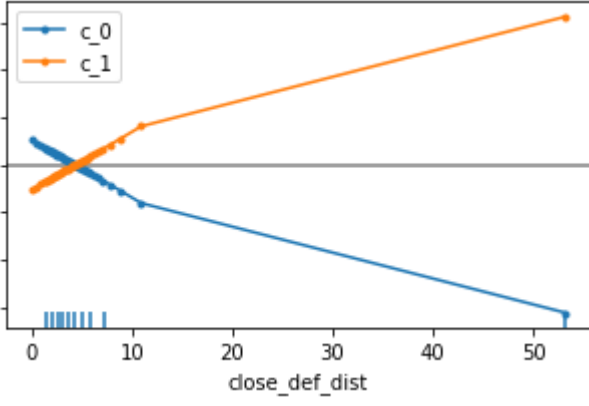
dribbles - At dribbles = 0 to 1, the model assigns negative probabilities towards classifying the instance as 1, and at dribbles >1, the probabilities become negative



touch time - At touch times between 0.1 to 1.8, the probabilities of classifying the instance as 1 is positive. It becomes negative at touch time >1.8



shot distance - at shot distance =0, the model assigns a 0.25 probability towards classifying the instance as a 1(made shot) and a -0.25 probability towards classifying the instance as a 0(missed shot). At a shot distance of ~15, the feature effect on the prediction is the same as the average feature effect. But for shot distances above 15, the model begins to assign negative probabilities towards classifying the instances as 1, and positive

	<p>probabilities towards classifying the instances as 0</p>
 <p>A line plot showing the Average Marginal Effect (ALE) on the y-axis (ranging from -0.02 to 0.05) against the variable 'pts_type' on the x-axis (ranging from 2.0 to 3.0). Two lines are plotted: a blue line for 'c_0' and an orange line for 'c_1'. The blue line starts at approximately 0.018 at pts_type=2.0 and decreases to approximately -0.018 at pts_type=3.0. The orange line starts at approximately -0.018 at pts_type=2.0 and increases to approximately 0.018 at pts_type=3.0. The two lines intersect at approximately pts_type=2.5, where the ALE is 0.00.</p>	<p>points type - for point type =2, the model assigns a positive probabilities towards classifying the instance as 1, and for point type=3, the model assigns positive probabilities towards classifying the instance as 0</p>
 <p>A line plot showing the Average Marginal Effect (ALE) on the y-axis (ranging from -0.02 to 0.05) against the variable 'close_def_dist' on the x-axis (ranging from 0 to 50). Two lines are plotted: a blue line for 'c_0' and an orange line for 'c_1'. The blue line starts at approximately 0.018 at close_def_dist=0 and decreases to approximately -0.018 at close_def_dist=50. The orange line starts at approximately -0.018 at close_def_dist=0 and increases to approximately 0.018 at close_def_dist=50. The two lines intersect at approximately close_def_dist=10, where the ALE is 0.00.</p>	<p>closest defender distance - Farther defender distances are assigned positive probabilities towards classifying the instance as 1</p>

Random Forest

ALE on Random Forest in probability space

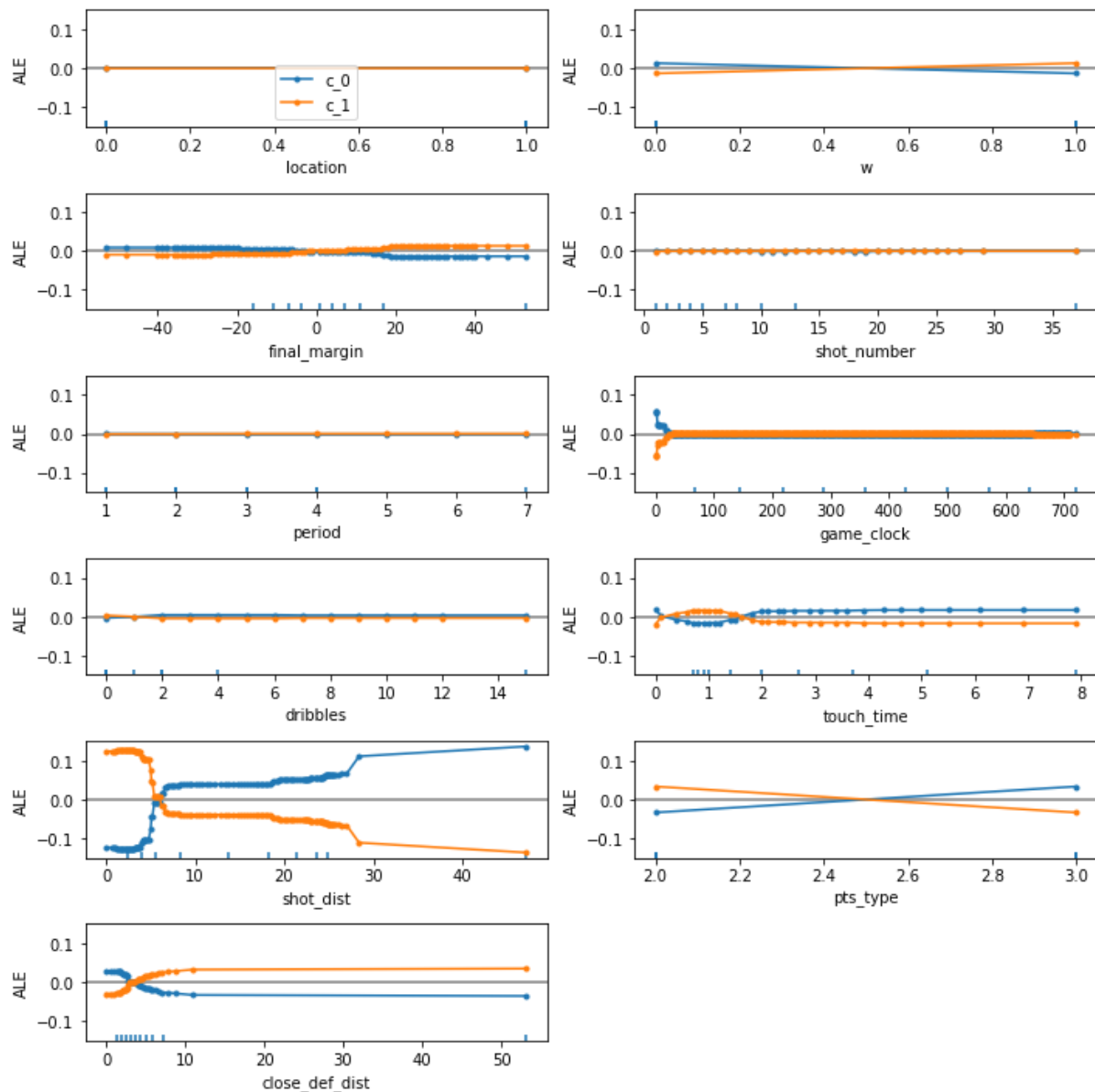


Figure 15: ALE on Random Forest Classifier

For the random forest model, location, shot number, period, and dribbles seem to have no effect on the average prediction. Shot distance seems to have the most effect on prediction. At shot distance between 0 and 5, the model assigns positive probabilities towards classifying an instance as one and negative assigned positive probabilities towards classifying an instance as 0. But for shot distances above 5, the model assigns higher probabilities towards classifying them as 0's. This means that, the closer a player is to the basket when the shot is made, the higher the probability that shot will be made.

Gradient Boosting

ALE on Gradient Boosting in logit space

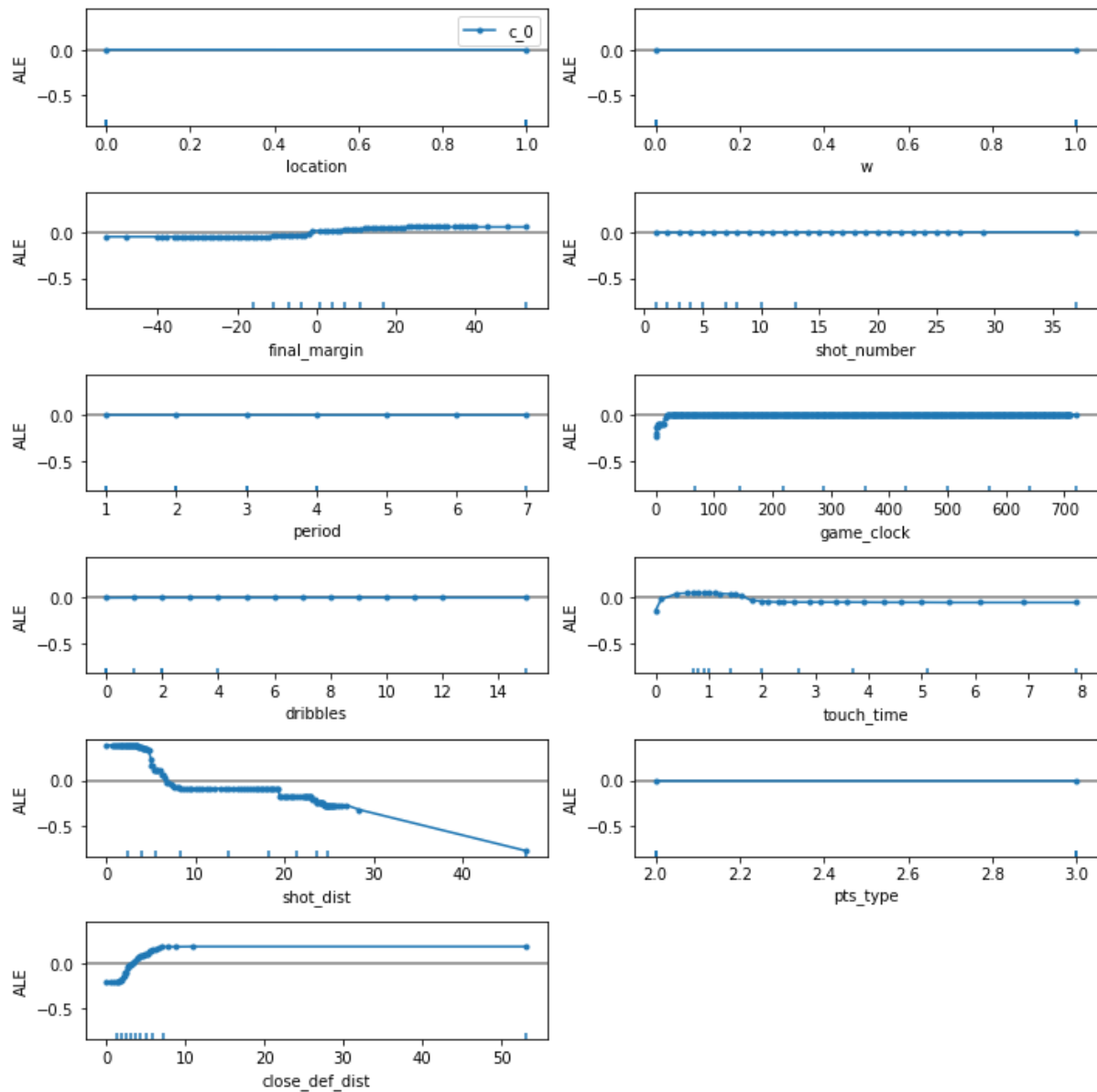


Figure 16: ALE on Gradient Boosting Classifier in logit space

ALE on Gradient Boosting in probability space

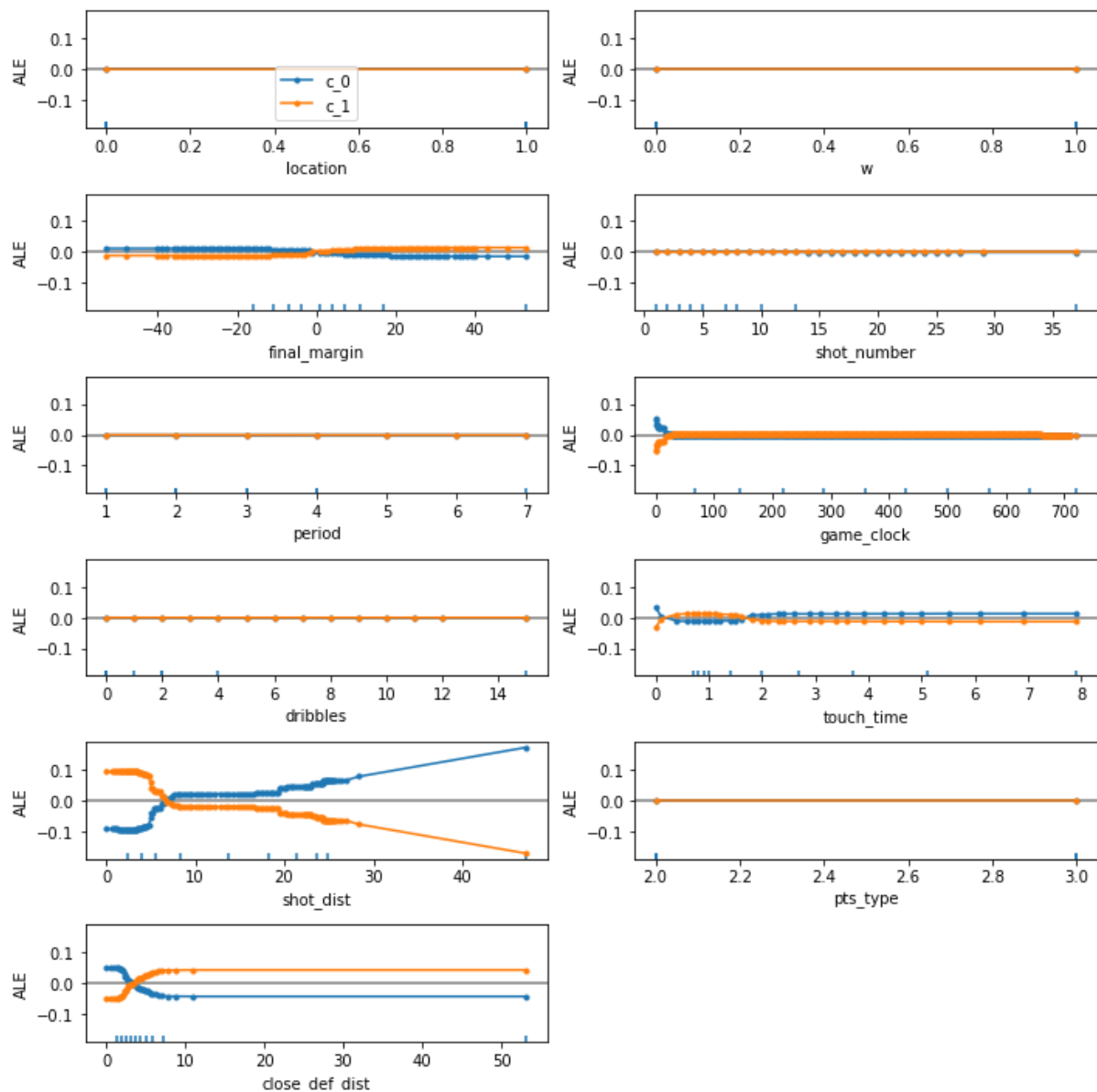


Figure 17: ALE on Gradient Boosting Classifier in probability space

This looks very similar to our random forest plots. With shot distance as the feature with the most impact

Feature Importance

Let's take a look at the feature importance on our gradient Boosting Classifier algorithm:

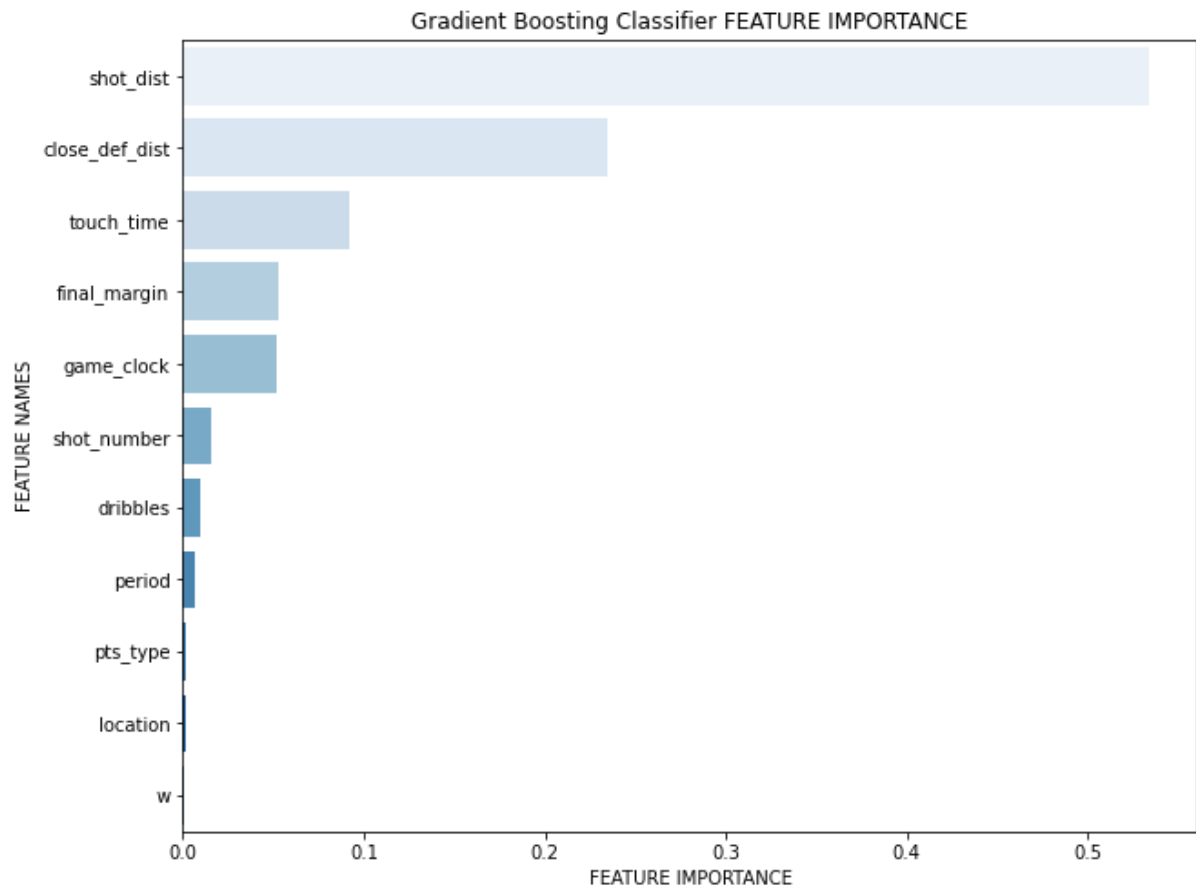


Figure 18: Gradient Boosting Classifier Feature Importance

Comparing the Effect of the feature, shot distance on Prediction across all three Models

Here, we examine the effect on shot distance, the feature with the highest importance on prediction for the three models trained for this analysis.



Figure 19: Effect of shot distance on Shots Missed



Figure 20: Effect of shot distance on Shots Made

All three models have similar effects in predicting missed and made shots respectively.

For missed shots, the models assigned higher probabilities to higher shot distances. While for made shots, the models assigned higher probabilities to smaller shot distances. This makes natural sense, as a player is more likely to score, if he's closer to the basket.

Outlier Detection

Outliers are observations that appear to deviate significantly from the other observations in a given data. The presence of the outer distribution instances on live data can lead to overconfident predictions and we do not want to act on these predictions. A good approach to check for outliers is the adoption of Alibi Detect.

Alibi Detect is an open-source Python library that focuses on outlier, adversarial and drifts detection. It runs on Seldon Core, an open-source platform to deploy machine learning models on Kubeflow at a massive scale.

Isolation Forest (IF)

These are tree-based models that run on the Alibi detect library, specifically used for outlier detection. The IF isolates observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature. The number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node. This path length averaged over a forest of random trees, is a measure of normality and is used to define an anomaly score. Outliers can typically be isolated quicker, leading to shorter paths. The algorithm is suitable for low to medium dimensional tabular data.

Result of Detecting Outliers with Isolation Forest

Let's assume we have some data which we know contains around 10% of outliers. We generated a batch of data with 10% outliers and detected the outliers in the batch.

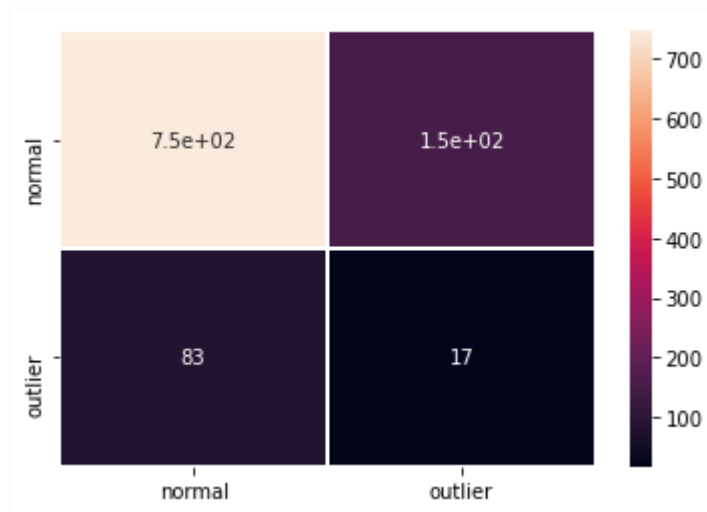


Figure 21: Confusion Matrix



Figure 22: Instance-level outlier vs. the outlier threshold

We can see that a lot of the outliers are below the threshold level. This was due to the misclassification of normal instances with the outliers by the model. This makes inferring a good threshold without explicit knowledge about the outliers difficult. Efforts to tune the model parameters for improvement didn't yield positive results.

Variable Auto- Encoder (AE)

The Variable Autoencoder is an outlier detection algorithm that also runs on the Alibi Detect library. The Auto-Encoder (AE) outlier detector is first trained on a batch of unlabeled, but normal (inlier) data. Unsupervised training is desirable since labelled data is often difficult to find. The AE detector tries to reconstruct the input it receives. If the input data cannot be reconstructed well, the reconstruction error is high and the data can be flagged as an outlier. The reconstruction error is measured as the mean squared error (MSE) between the input and the reconstructed instance.

Results of Detecting Outliers Using V-AE

Let's assume we have some data which we know has ~ 10% of outliers. We generated a batch of data with 10% outliers and detected the outliers in the batch.

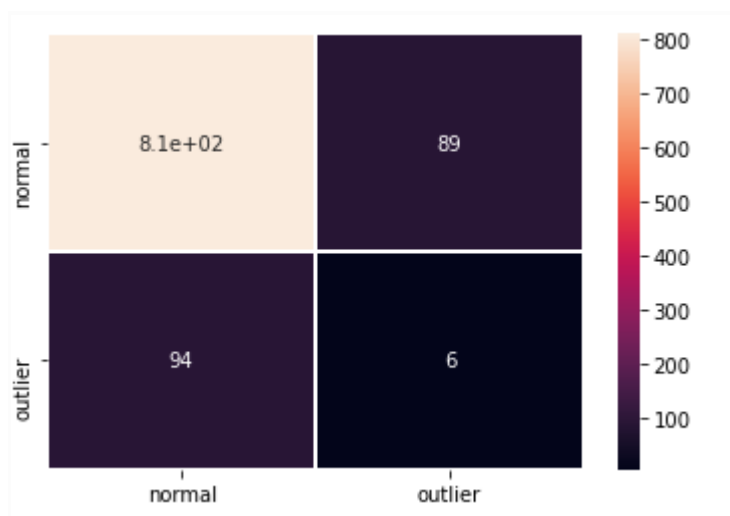


Figure 23: Confusion matrix

```
plot_instance_score(od_preds, y_outlier, labels, od.threshold)
```

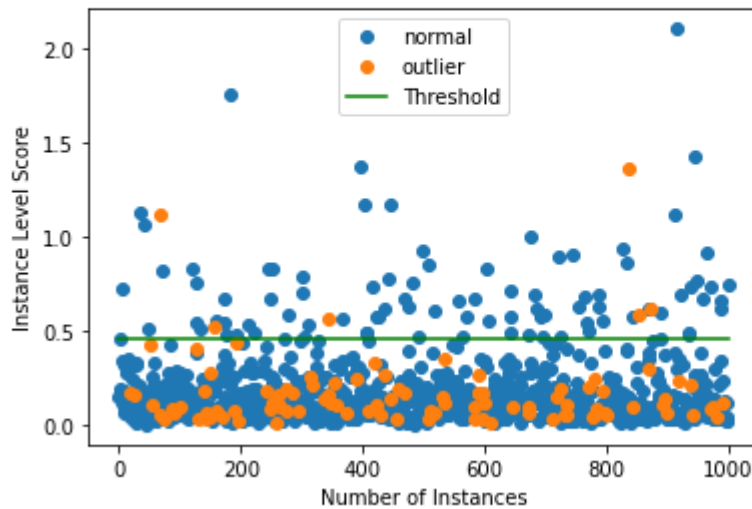


Figure 24: Instance-level outlier vs. the outlier threshold

We can see that most of the outliers lie below the threshold. This was due to the misclassification of the outliers with the normal instances of the data by the model. Efforts were made to tune the model to improve the classification performance but didn't yield positive results.

From the foregoing, we can conclude that our two outlier detection models slightly captured some outliers above the threshold level and misclassified the rest with the normal instances of the data. This could be an indicator of a low predictive ability of the predictors in the dataset.

Model Deployment using Kubeflow Platform

Our model has been created, model explainability and outlier detection of the data has been taken into account, our next step involves deploying this workflow where it can be used to produce predictions of new NBA data. To do this, we utilized the Kubeflow platform. Kubeflow provides end to end functionality for executing ML

projects. It runs on the Kubernetes engine which automates deployment, scaling, and management of containerized applications.

In the data science industry, it is given that a model is only really useful when it has been put into production, however, the production is often rife with obstacles based on portability and scalability.

Kubeflow alleviates some of these obstacles, making it easier to develop, deploy, and manage portable, scalable machine learning projects, with tools that support the full lifecycle of the project, including ideation and iteration via Jupyter notebooks. With a relatively easy to use process, it reduces the barrier to entry for developing and maintaining machine learning products, with a single, unified platform for running tasks like data ingestion, transformation, and analysis, model training, evaluation, and serving, as well as monitoring and logging. As a portable platform, Kubeflow can be run either locally, on-premise, or on the cloud platform of your choice. As it is backed by Kubernetes, scalability can be leveraged for all aspects and components of the project.

Our first step in deploying our project involves building a pipeline for our machine learning code. Kubeflow allows for lightweight and reusable types of components, with the lightweight components being useful for fast and iterative development. In this project, we used reusable components, this involves creating containerized functions of all of our ML code which could then be pulled by our pipeline.

The process is as follows:

1. Creation of self-contained python functions that will pass the needed data between themselves.

```
import argparse
def gbc(clean_data):
    import joblib
    import numpy as np
    import pandas as pd
```

```

from sklearn import metrics
from sklearn.ensemble import GradientBoostingClassifier

data = joblib.load(clean_data)
X_train = data['X_train']
y_train = data['Y_train']
X_test = data['X_test']
y_test = data['Y_test']

gbcla = GradientBoostingClassifier(max_features='auto',
                                   n_estimators=100, random_state=42, max_depth=5,
min_samples_leaf=100, learning_rate = 0.08)

gbcla.fit(X_train, y_train)

y_pred = gbcla.predict(X_test)

# Test score
test = gbcla.score(X_test, y_test)
print('test accuracy:')
print(test)
train = gbcla.score(X_train, y_train)
print('train accuracy:')
print(train)

#Classification Report
report = metrics.classification_report(y_test, y_pred,
output_dict=True)
df_classification_report = pd.DataFrame(report).transpose()
print(df_classification_report)
gbc_metrics = {'train': train, 'test':test,
'report':df_classification_report}
joblib.dump(gbc_metrics, 'gbc_metrics')

if __name__ == '__main__':
    parser = argparse.ArgumentParser()

```

```
parser.add_argument('--clean_data')
args = parser.parse_args()
gbc(args.clean_data)
```

The above image shows the python code for our Gradient Boosting Classifier, each package needed for the function to run independently is imported within the function. It collects clean data from our preprocessing step as an input and outputs the evaluation metrics for the model. To containerize this function we will need to write a Dockerfile which will provide the container with the needed environment requirements and directory instructions.

```
FROM python:3.8
WORKDIR /gbc
RUN pip install -U scikit-learn numpy pandas joblib
COPY gradient.py /gbc
ENTRYPOINT [ "python", "gradient.py" ]
```

This provides the container with a python 3.8 base, directs the creation of a 'gbc' working directory then installs using the pip package manager the required python packages. The last two lines direct the files needed by the container and how the code is executed.

With our function, gradient.py, and the dockerfile in the same directory, we can build a docker image containing the function. An account with an image repository like DockerHub is needed to store the image where it can be called from the cloud.

```
# build the image
docker build --tag=gbc_nba:v.0.1
# tag to a docker repository
```

```
docker tag gbc_nba:v.0.1 mavencodvv/gbc_nba:v.0.1
# push the image to the repository
docker push mavencodvv/gbc_nba:v.0.1
```

As these docker commands are run in the same directory from the command line, the image is built from our files and pushed to the repository. These steps are taken for each component of our machine learning code, once this is done we can create our pipeline using Kubeflow's software development kit in a notebook.

```
# Installing the Kubeflow SDK
!python -m pip install --user --upgrade pip
!pip3 install kfp --upgrade --user

# Restart the runtime then import the packages
import kfp
from kfp import dsl
import kfp.components as comp
```

2. With the packages installed and imported, we can build pipeline components using Kubeflow's unique codebase. Each component is defined using the image we created and the inputs and outputs are given as arguments of a function.

```
def gbc_op(clean_data):
    return dsl.ContainerOp(
        name = 'GBC',
        image = 'mavencodvv/gbc_nba:v.0.2',
        arguments = ['--clean_data', clean_data
                    ],
        file_outputs={
            'gbc_metrics': '/gbc/gbc_metrics'
        }
    )
```

Once each component is defined, we can create our pipeline, explicitly setting each parameter that is needed.

```
@dsl.pipeline(
    name='Shot Result Prediction',
    description='An ML reusable pipeline that predicts whether a shot
from an NBA player will go in or not'
)

# Define parameters to be fed into pipeline
def nba_pipeline():

    _load_data_op = load_data_op()

    _preprocess_op = preprocess_op(
dsl.InputArgumentPath(_load_data_op.outputs['data'])).after(_load_data_op)

    _rf_op = rf_op(
dsl.InputArgumentPath(_preprocess_op.outputs['clean_data'])).after(_preprocess_op)

    _gbc_op = gbc_op(
dsl.InputArgumentPath(_preprocess_op.outputs['clean_data'])).after(_preprocess_op)

    _lr_op = lr_op(
dsl.InputArgumentPath(_preprocess_op.outputs['clean_data'])).after(_preprocess_op)

    _results_op = results(
```

```

        dsl.InputArgumentPath(_rf_op.outputs['rf_metrics']),
        dsl.InputArgumentPath(_gbc_op.outputs['gbc_metrics']),

dsl.InputArgumentPath(_lr_op.outputs['lr_metrics'])).after(_rf_op, _gbc_op, _lr_op)

```

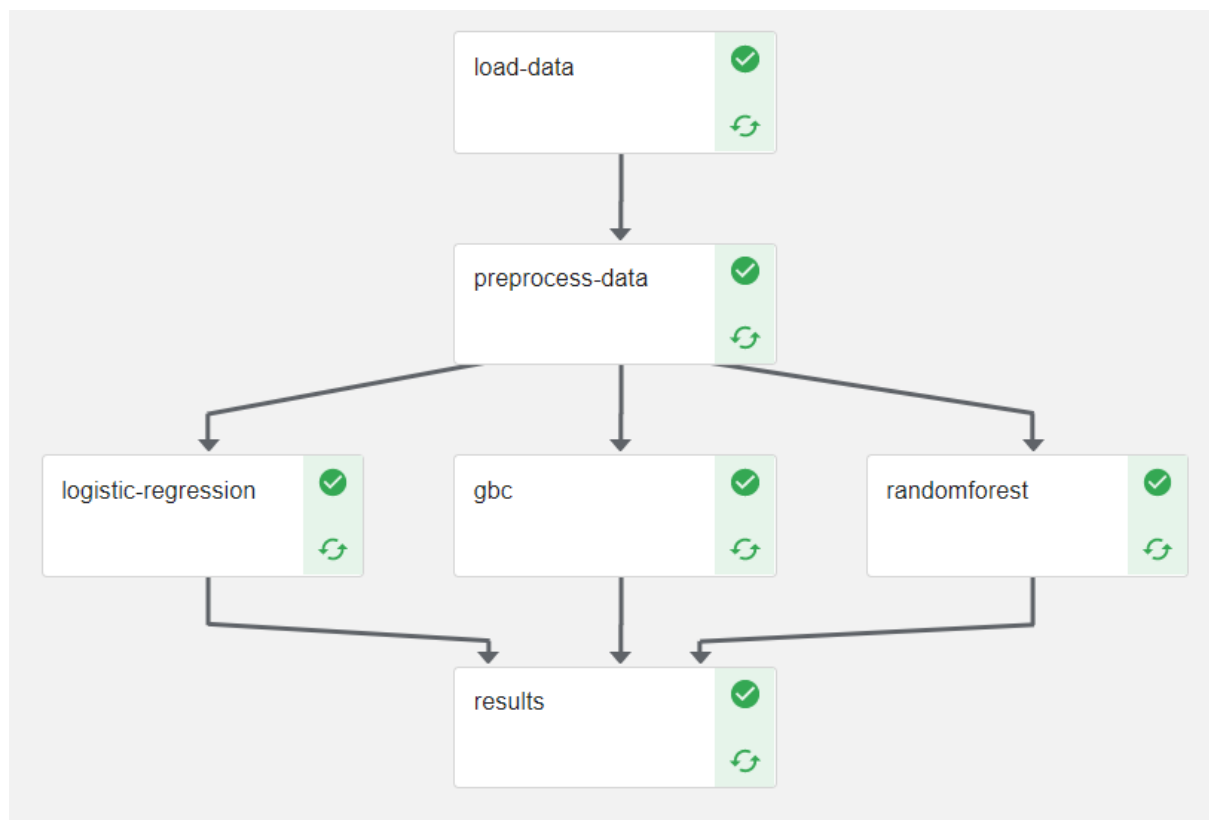
3. Once this is done we can compile this function, Yaml, zip or tar.gz are acceptable formats, and upload it on the Kubeflow platform using the `kfp.Client` class or a manual upload.

```

# Compile pipeline to generate compressed YAML definition of the
pipeline.
experiment_name = 'nba_pipeline'
kfp.compiler.Compiler().compile(nba_pipeline,
'{}.yaml'.format(experiment_name))

# Client requires your Kubeflow endpoint and credentials
client = kfp.Client()
client.create_run_from_pipeline_func(nba_pipeline, arguments={})

```

Each model in the pipeline is trained and evaluated with the accuracy on train and test data calculated as well as the precision, recall and f1 scores of the predicted classes. These metrics are compiled and exported in the results component.

Gradient Boosting Classifier

'Classification report':

	precision	recall	f1-score	support
0	0.617475	0.835132	0.709997	19640.000000
1	0.652799	0.374669	0.476090	16249.000000
accuracy	0.626654	0.626654	0.626654	0.626654
macro avg	0.635137	0.604901	0.593044	35889.000000
weighted avg	0.633468	0.626654	0.604094	35889.000000,

'Test accuracy': 0.6266544066427039,

'Train accuracy': 0.6293288750895629

Summary

In this project, we explored the 2014-2015 season, looking at the patterns in scoring showing peaks in scoring frequency around the middle of the season. The 2-pointer being the most frequent shot selection and time spent with the ball before was not a significant influence on shot selection on average. We also looked at the top 10 best defenders and shooters, after highlighting Stephen Curry the MVP of the season as a prolific scorer, we explored some quantitative differences in his shot selection and playing style.

We used Machine Learning Algorithms to predict whether a shot would be made or missed. Our model predicted shot results at a 62.67% accuracy.

To better understand our results, we used the ALE (Accumulated Local Effects) explainer to explain the effect of each feature in our prediction. After dropping *points* and *field goals made* features, which both had a correlation of 1 with our target feature (*shot results*), our model explainer showed *shot distance* to be the most important feature in our prediction.

To check for possible outliers, we used two outlier detection models: Isolation Forest and Variable Auto-Encoder, to identify outliers in a batch of the dataset. Our two outlier detection models slightly captured some outliers above the threshold level and misclassified the rest with the normal instances. This could be an indicator of a low predictive ability of the predictors in the dataset.

Conclusion

The sporting world is no stranger to data analytics, with each season generating scores of records and statistics of performances of players, the managing staff and even the fans. However, the utilization of data in new ways and new use cases have grown in recent times as examples and cases of teams boosting their performance with the power of analytics and data science.

Data science concepts have been seen to help with preparation before, during and after the match with use cases in health and conditioning of the players, management of the team and fans as well as talent scouting.

A model is only as good as the quality of data it is trained. We need no crystal ball to 'predict' that the performance of our best model could be better. FurtherThe endpoint of an ML pipeline is only as good as the quality of data input at the start.

References

<https://www.kaggle.com/dansbecker/nba-shot-logs>

<https://www.enjine.com/blog/interpreting-machine-learning-models-accumulated-local-effects/>

<https://christophm.github.io/interpretable-ml-book/ale.html>

<https://github.com/SeldonIO/alibi>

https://docs.seldon.io/projects/alibi-detect/en/latest/examples/od_if_kddcup.html

https://docs.seldon.io/projects/alibi-detect/en/latest/examples/od_vae_kddcup.html

[https://en.wikipedia.org/wiki/Logit#:~:text=In%20statistics%2C%20the%20logit%20\(%2F,.](https://en.wikipedia.org/wiki/Logit#:~:text=In%20statistics%2C%20the%20logit%20(%2F,.)