# Cluster Analysis for Fantasy Basketball Draft Trade Strategy

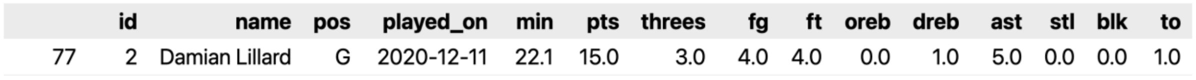
*Authors: Carl Ausmees, Troy Jennings*

## 1. Introduction

In this study, we will answer the following questions: what comparisons can we make between clusters through a K-Means algorithm vs. a DBSCAN algorithm, and are these methods suitable for comparing player similarity in the NBA (National Basketball Association) based on individual player in-game statistics? The dataset for this project will include individual per-game player statistics from the National Basketball Association (NBA). This dataset was aggregated and cleansed by members of the project team for a tangential project.

## 2. Background

Fantasy sports leagues enable fans ("**commissioners**") to put together a virtual team of real-life players, to compete against other virtual teams in a variety of formats, based on the players' actual game statistics. In fantasy sports, the objective is to assemble a team of NBA players through a mock draft. These teams accumulate fantasy points across categories over the course of a season, determining the virtual team's rank in each category. The final standings of the fantasy teams at the end of the season determine the league winner. Players' statistics and other metrics will be evaluated in this project including the following: player position, minutes played, points, three-points made, field goals made, free throws made, offensive/defensive rebounds, assists, steals, blocks, turnovers, and date played on. The idea throughout this project will be to evaluate players statistics through clustering. Categorical statistics in each category are described in **Table 1 – Dataframe and basketball statistics explained** in the **Appendix.**



| | id | name | pos | played_on | min | pts | threes | fg | ft | oreb | dreb | ast | stl | blk | to |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 77 | 2 | Damian Lillard | G | 2020-12-11 | 22.1 | 15.0 | 3.0 | 4.0 | 4.0 | 0.0 | 1.0 | 5.0 | 0.0 | 0.0 | 1.0 |

**Figure 1 –** Example of a player's game statistics from 12-11-2020.

## 3. Exploratory Analysis & Problem Statement Formulation

Based on exploratory analysis, it is immediately apparent that players' performance relies on multiple factors, as in any sport. Some players produce more equally and consistently between all the categories, while others exhibit more variability with specialized skillsets. This adds an interesting wrinkle in decision-making when fantasy team owners initially draft the players, and more importantly, how to approach a player trade through a season: how do different forms of clustering help compare player similarity based on stats? Can we effectively determine whether a trade made by two members of a league is "fair," not worth it, or in your best interest? To tackle this question, we aim to classify players into an optimal number of clusters (K-Means) or based on optimal epsilon (DBSCAN), plot the most optimal clusters, and evaluate the created clusters across desired statistics to determine if we can evaluate a player trade while comparing the different created clusters.

## 4. Description of Data

Given we approach the problem from a clustering perspective, all features are considered inputs with no ground truth for cluster assignment. Most of the input variables are numerical apart from the player identifier variables – "player_ id", "first_name", and "last_name" – and the categorical "position" variable. A full description of the dataset variables and their datatypes can be found in **Table 1 – Dataframe and Basketball Statistics Explained** table in the **Appendix** section.

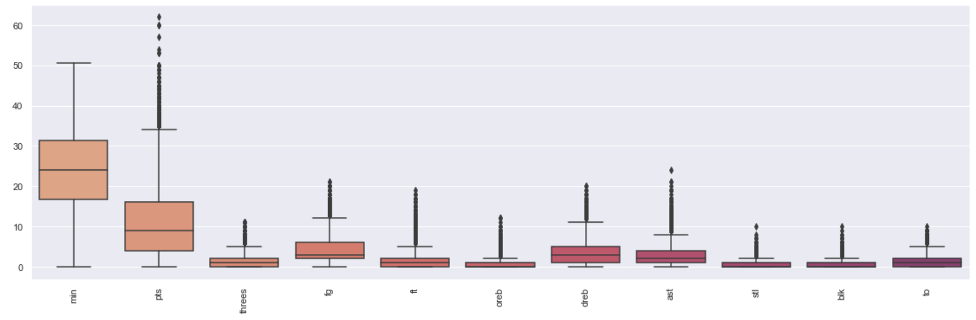


**Figure 2 –** Boxplots for all post-cleaning numerical features.

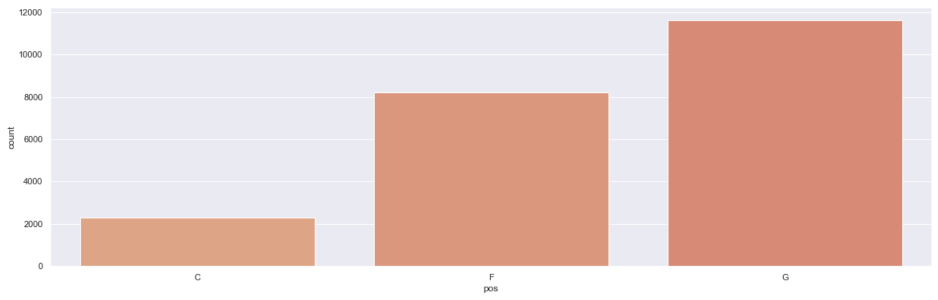


**Figure 3 –** Countplots for all post-cleaning categorical features (position).

## 5. Data Preprocessing and Preparation

The original data was gathered and cleansed from tangential project work, so the major data preprocessing steps were minimal for the project at hand. To get the data in a form conducive for analysis we conducted multiple steps. First, drop extraneous features "player_statistic_id", "player_id-2", "fixture_id", "is_starter", "created_at", "threes_attempted", "field_goals_attempted", and "free_throws_attempted" since they represent features that are extraneous. Secondly, convert "player_status" to a binary categorical to facilitate filtering out observations (I.e., games) where a player did not play in the game and therefore did not have any values. Next, convert "played_on" to a true datetime format to facilitate filtering for observations that occurred on or before a specified date and recode "position_primary" to 3 values – "G", "F", or "C". This was conducted to bin every player into distinct groups since players commonly play multiple positions but to simplify one-to-one player analysis, distinct groups are preferred. It's necessary to then convert "seconds_played" to minutes as a "pre-transformation" for exploratory data analysis on more similar scales across all features and consolidate each player's "first_name" and "last_name" features into a single feature – "name" – to facilitate player identifier uniqueness. Lastly, cast categorical features – "position_primary" and "player_id" – to true categorical features to avoid them being used in feature scaling and rename all columns to more concise terms for easier sub-setting or selection. No data partitioning into training and testing was necessary since the problem space was clustering.

## 6. Research Methodology & Implementation

To simplify the analysis for addressing our research questions, the data was filtered down to a subset by selecting a random player (I.e., the "trading player") from all available players and choosing a random date from all "played_on" dates (I.e., the "trade date"). This filtering process mimics the player trade process and results in a dataset that includes only players of the same position as the trading player and only observations that have a "played_on" date on-or-before the trade date. This process simplifies the analysis to focus on cross-cluster analysis of players of the same position.

Once the dataset has been filtered down, the cumulative sum of all numeric features (I.e., player statistics) is aggregated by player to reduce each player's statistics to a single row for all games played up until the trade date. These numerical sums are then normalized using a standard scaler to normalize the data into a smaller range for modeling.

Because this project operates in an unsupervised learning environment, we begin the modeling process by searching for the **optimal number of components** for dimensionality reduction, the **optimal number of clusters** for the KMeans algorithm, and the **optimal epsilon value** for the DBSCAN algorithm. The code and plot results of this initial investigation for optimal parameters are shown in **Figure 7** through **Figure 12** in the **Appendix**.

Once the optimal parameters are found, a model pipeline is built to streamline the coding and execution of the final models with their respective optimal parameters. It is important to note that no further hyperparameter tuning was done since the problem space is unsupervised learning. The modeling step also adds the predicted cluster assignments, for both the KMeans and DBSCAN algorithms, to the summarized dataframe as shown **Figure 16** in the **Appendix**. These cluster assignments are then used to gauge whether each clustering algorithm helps address our research question.

## 7. Results & Conclusion

After predicting cluster assignments for each player in the filtered dataset, we plot each player statistic feature across clusters for both KMeans and DBSCAN. The full plots for that assessment are shown in **Figure 14** and **Figure 15** in the **Appendix** but we show samples from those plots for the "minutes_played" feature.

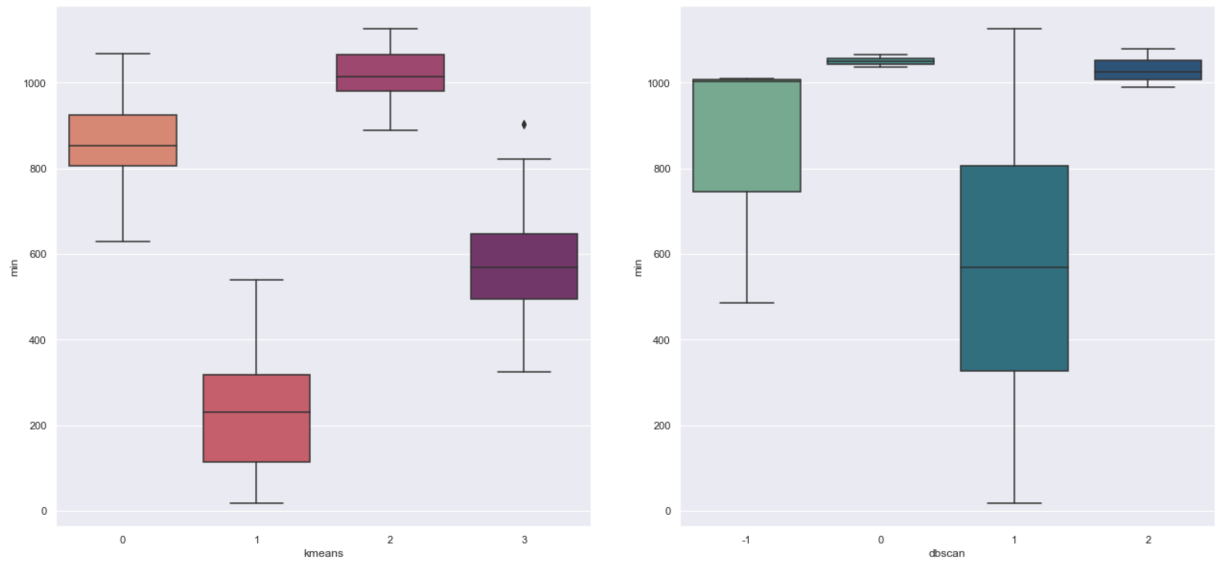


**Figure 4 –** Sample boxplots for the minutes_played feature.

What we immediately see in both plots above is that both algorithms objectively group players into distinct clusters, assuming we consider the outliers from DBSCAN as its own cluster. What these plots do not explain is the distributions within clusters for both algorithms. We can analyze the clusters further with bar charts shown below.

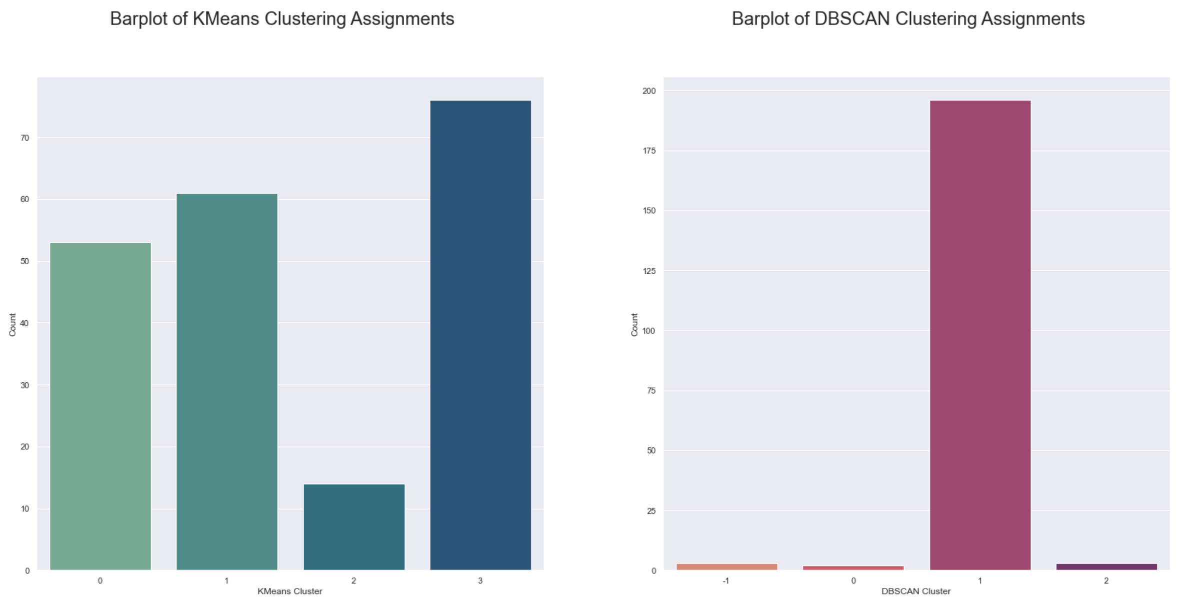


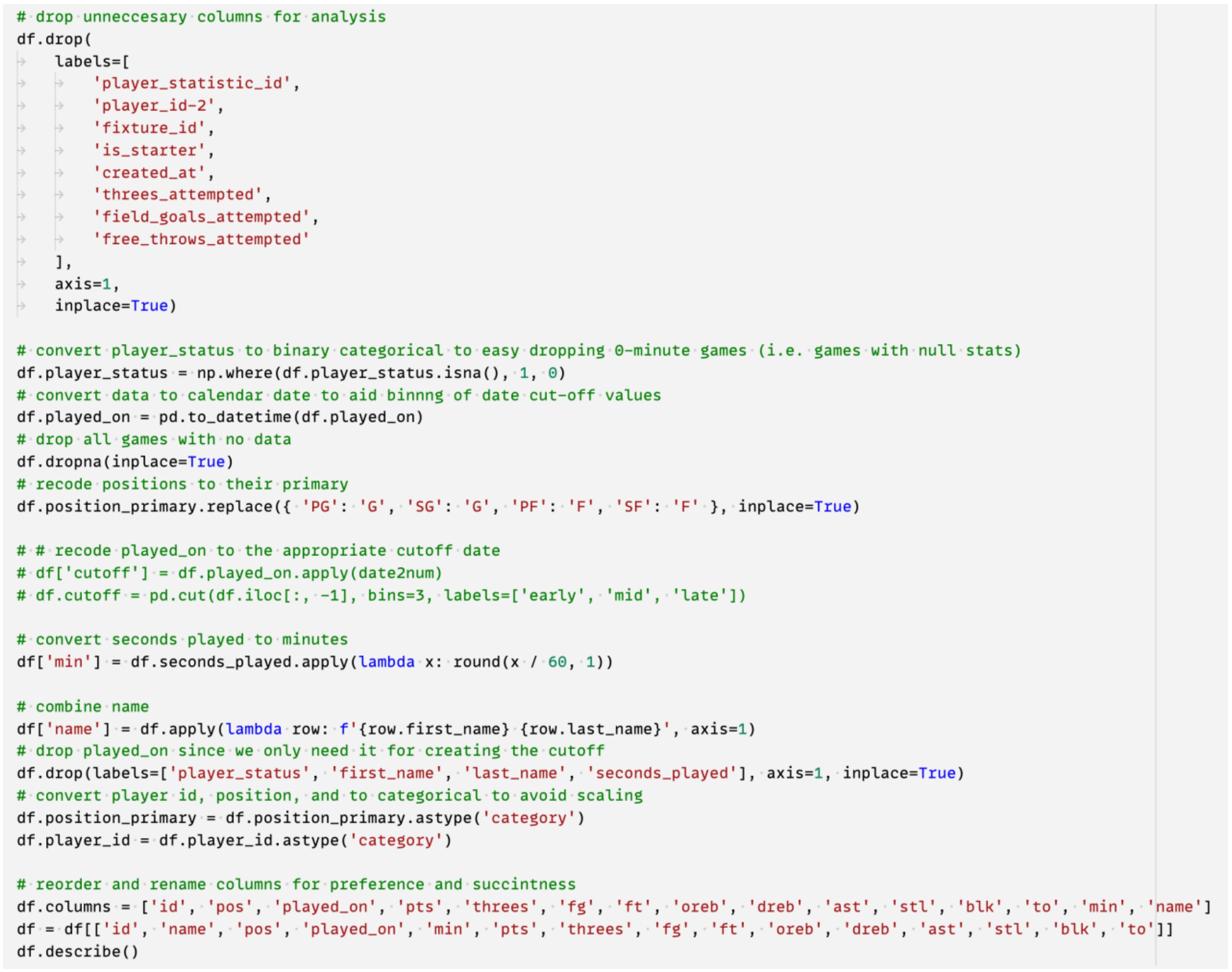
**Figure 5 –** Bar charts for the cluster distributions.

From the bar charts, we see that DBSCAN is clustering almost all players into a single cluster. Upon further inspection, we found that there were only a couple of players in each of the clusters 0 and 2 and the –1 outlier group. What this indicated to us was that DBSCAN does not work well on this dataset and, further, that it doesn't capture the inherent variance in the dataset. From this we would conclude that the KMeans algorithm is suitable for comparing player similarity in the NBA based on individual player in-game statistics, whereas DBSCAN is not. Using the KMeans model, we could then make an evaluation of a trade by comparing the mean statistics of the cluster to which the trade player belongs and to which the proposed player belongs. From this, if the proposed player is in the same cluster as the trade player or a cluster with higher mean statistics, then the trade would be considered “fair” or even in your best interest. Conversely, if the proposed player is in a cluster with lower mean statistics, then the trade would be considered “not worth it”.

## 8. Appendix

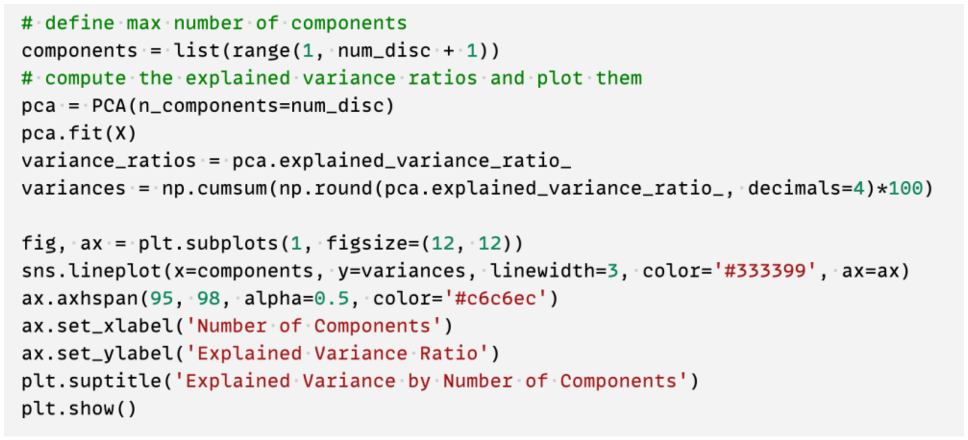
| Column | Type | Description |
|---|---|---|
| **Player ID** | Numeric | Unique identifier for each player |
| **First Name** | String | Player first name |
| **Last Name** | String | Player last name |
| **Position Primary** | Categorical (String) | Player primary position played |
| **Seconds Played** | Numeric | Seconds played (in a game) |
| **Points** | Numeric | Points scored (in a game) |
| **Threes Attempted** | Numeric | Three-point shots attempted (in a game) |
| **Threes Made** | Numeric | Three-point shots made (in a game) |
| **Field Goals Attempted** | Numeric | Field goals attempted (in a game) |
| **Field Goals Made** | Numeric | Field goals made (in a game) |
| **Free Throws Attempted** | Numeric | Free throws attempted (in a game) |
| **Free Throws Made** | Numeric | Free throws made (in a game) |
| **Offensive Rebounds** | Numeric | Offensive rebounds (in a game) |
| **Defensive Rebounds** | Numeric | Defensive rebounds (in a game) |
| **Assists** | Numeric | Assists (in a game) |
| **Steals** | Numeric | Steals made (in a game) |
| **Blocks** | Numeric | Blocks (in a game) |
| **Turnovers** | Numeric | Turnovers (in a game) |

**Table 1 –** Dataframe and basketball statistics explained.



```python
# drop unneccesary columns for analysis
df.drop(
    labels=[
        'player_statistic_id',
        'player_id-2',
        'fixture_id',
        'is_starter',
        'created_at',
        'threes_attempted',
        'field_goals_attempted',
        'free_throws_attempted'
    ],
    axis=1,
    inplace=True)

# convert player_status to binary categorical to easy dropping 0-minute games (i.e. games with null stats)
df.player_status = np.where(df.player_status.isna(), 1, 0)
# convert data to calendar date to aid binnng of date cut-off values
df.played_on = pd.to_datetime(df.played_on)
# drop all games with no data
df.dropna(inplace=True)
# recode positions to their primary
df.position_primary.replace({ 'PG': 'G', 'SG': 'G', 'PF': 'F', 'SF': 'F' }, inplace=True)

# # recode played_on to the appropriate cutoff date
# df['cutoff'] = df.played_on.apply(date2num)
# df.cutoff = pd.cut(df.iloc[:, -1], bins=3, labels=['early', 'mid', 'late'])

# convert seconds played to minutes
df['min'] = df.seconds_played.apply(lambda x: round(x / 60, 1))

# combine name
df['name'] = df.apply(lambda row: f'{row.first_name} {row.last_name}', axis=1)
# drop played_on since we only need it for creating the cutoff
df.drop(labels=['player_status', 'first_name', 'last_name', 'seconds_played'], axis=1, inplace=True)
# convert player id, position, and to categorical to avoid scaling
df.position_primary = df.position_primary.astype('category')
df.player_id = df.player_id.astype('category')

# reorder and rename columns for preference and succintness
df.columns = ['id', 'pos', 'played_on', 'pts', 'threes', 'fg', 'ft', 'oreb', 'dreb', 'ast', 'stl', 'blk', 'to', 'min', 'name']
df = df[['id', 'name', 'pos', 'played_on', 'min', 'pts', 'threes', 'fg', 'ft', 'oreb', 'dreb', 'ast', 'stl', 'blk', 'to']]
df.describe()
```

**Figure 6 –** Data cleaning code.



```python
# define max number of components
components = list(range(1, num_disc + 1))
# compute the explained variance ratios and plot them
pca = PCA(n_components=num_disc)
pca.fit(X)
variance_ratios = pca.explained_variance_ratio_
variances = np.cumsum(np.round(pca.explained_variance_ratio_, decimals=4)*100)

fig, ax = plt.subplots(1, figsize=(12, 12))
sns.lineplot(x=components, y=variances, linewidth=3, color='#333399', ax=ax)
ax.axhspan(95, 98, alpha=0.5, color='#c6c6ec')
ax.set_xlabel('Number of Components')
ax.set_ylabel('Explained Variance Ratio')
plt.suptitle('Explained Variance by Number of Components')
plt.show()
```

**Figure 7 –** Code for finding optimal components in PCA.

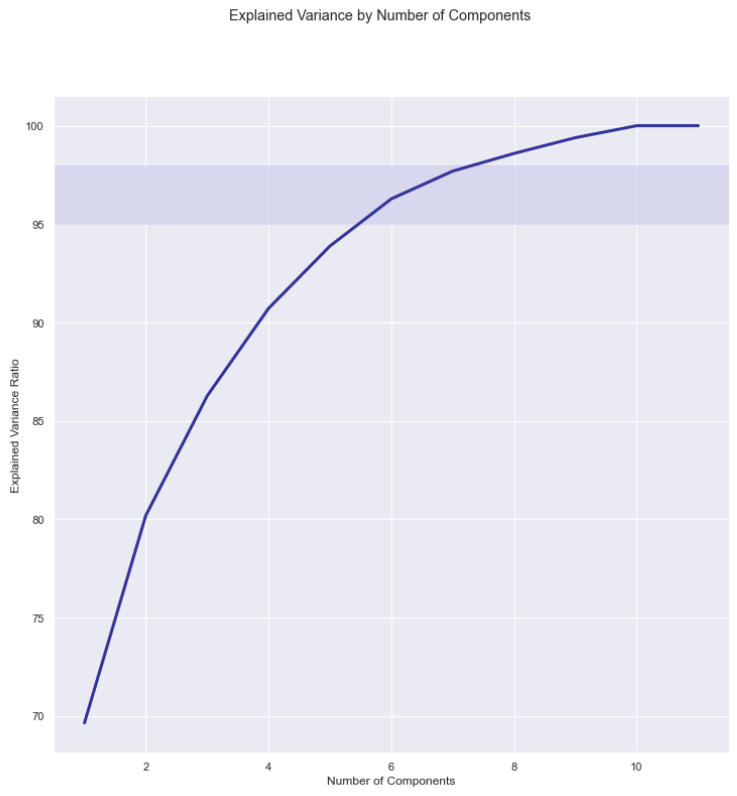
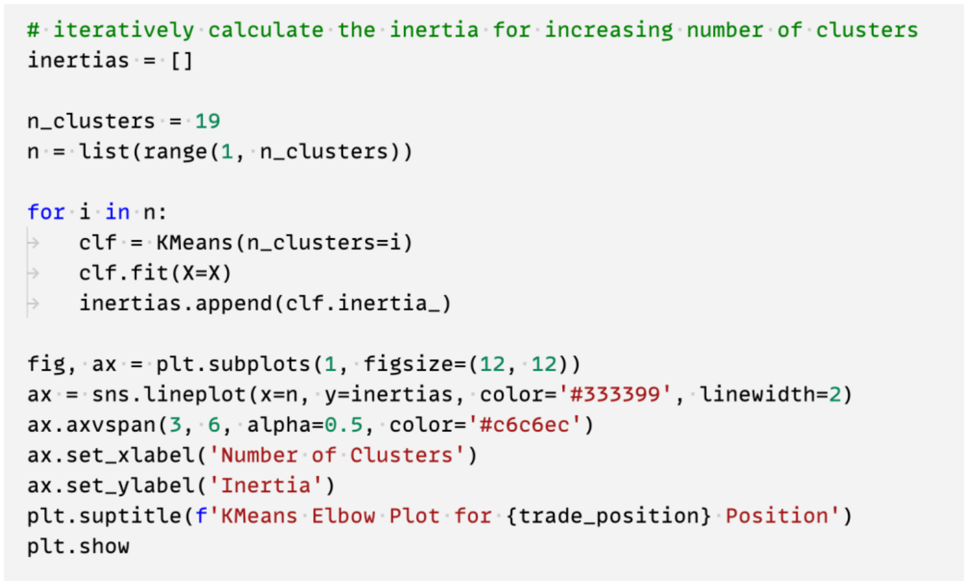


**Figure 8 –** Explained variance by increasing number of components plot.



```
# iteratively calculate the inertia for increasing number of clusters
inertias = []

n_clusters = 19
n = list(range(1, n_clusters))

for i in n:
    clf = KMeans(n_clusters=i)
    clf.fit(X=X)
    inertias.append(clf.inertia_)

fig, ax = plt.subplots(1, figsize=(12, 12))
ax = sns.lineplot(x=n, y=inertias, color='#333399', linewidth=2)
ax.axvspan(3, 6, alpha=0.5, color='#c6c6ec')
ax.set_xlabel('Number of Clusters')
ax.set_ylabel('Inertia')
plt.suptitle(f'KMeans Elbow Plot for {trade_position} Position')
plt.show
```

**Figure 9 –** Code for finding optimal cluster in KMeans.

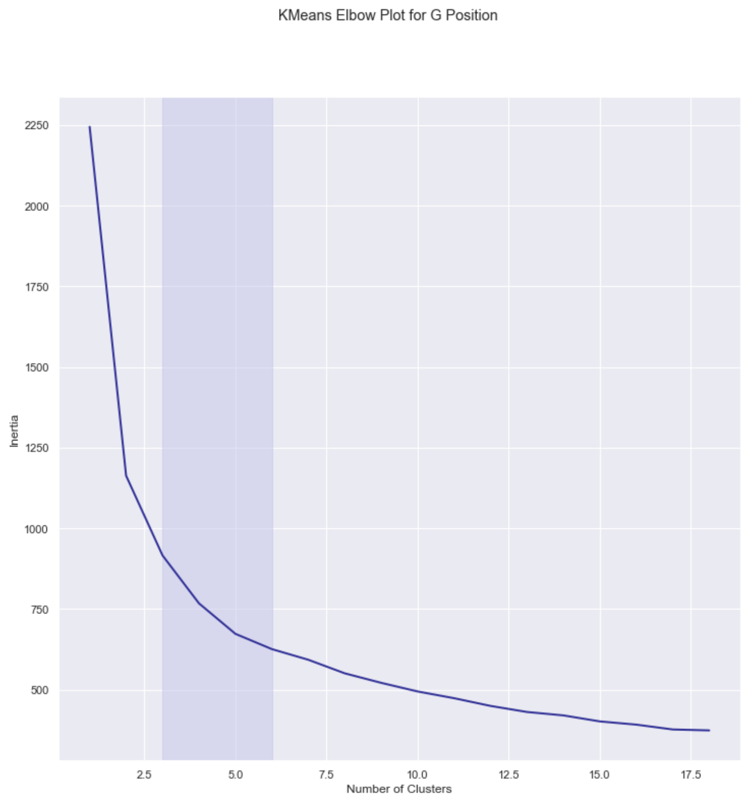
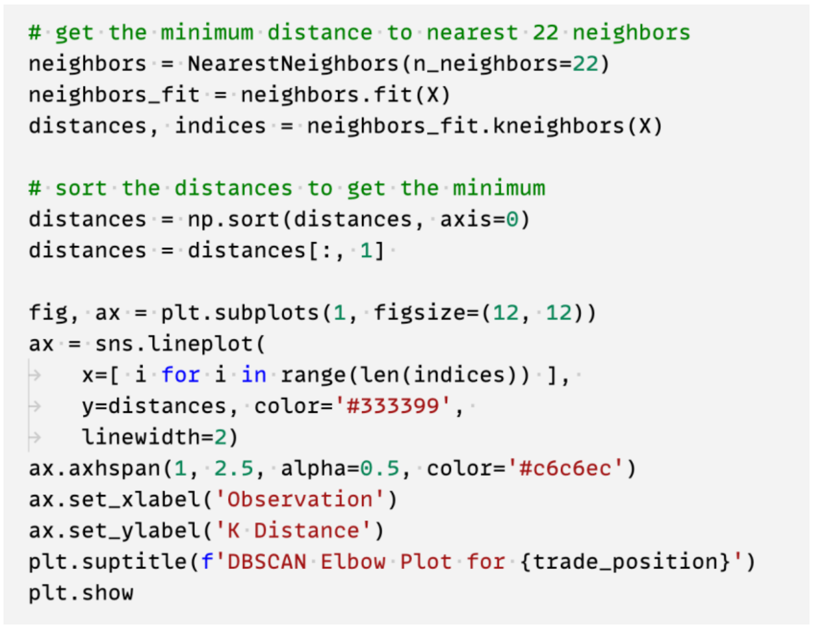


**Figure 10 –** Inertia by increasing number of clusters plot.



```
# get the minimum distance to nearest 22 neighbors
neighbors = NearestNeighbors(n_neighbors=22)
neighbors_fit = neighbors.fit(X)
distances, indices = neighbors_fit.kneighbors(X)

# sort the distances to get the minimum
distances = np.sort(distances, axis=0)
distances = distances[:, 1]

fig, ax = plt.subplots(1, figsize=(12, 12))
ax = sns.lineplot(
    x=[ i for i in range(len(indices)) ],
    y=distances, color='#333399',
    linewidth=2)
ax.axhspan(1, 2.5, alpha=0.5, color='#c6c6ec')
ax.set_xlabel('Observation')
ax.set_ylabel('K Distance')
plt.suptitle(f'DBSCAN Elbow Plot for {trade_position}')
plt.show
```

**Figure 11 –** Code for finding optimal epsilon value (eps) in DBSCAN.

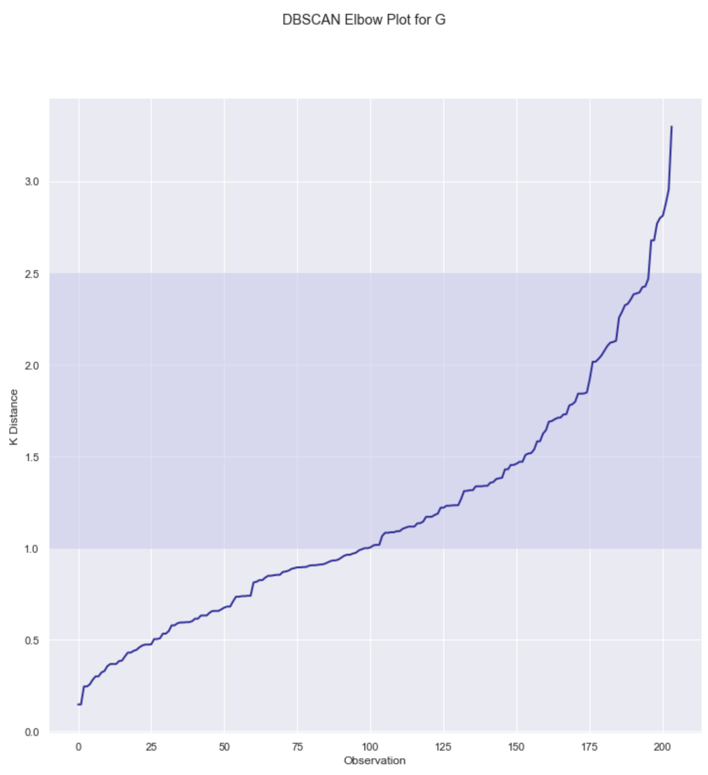
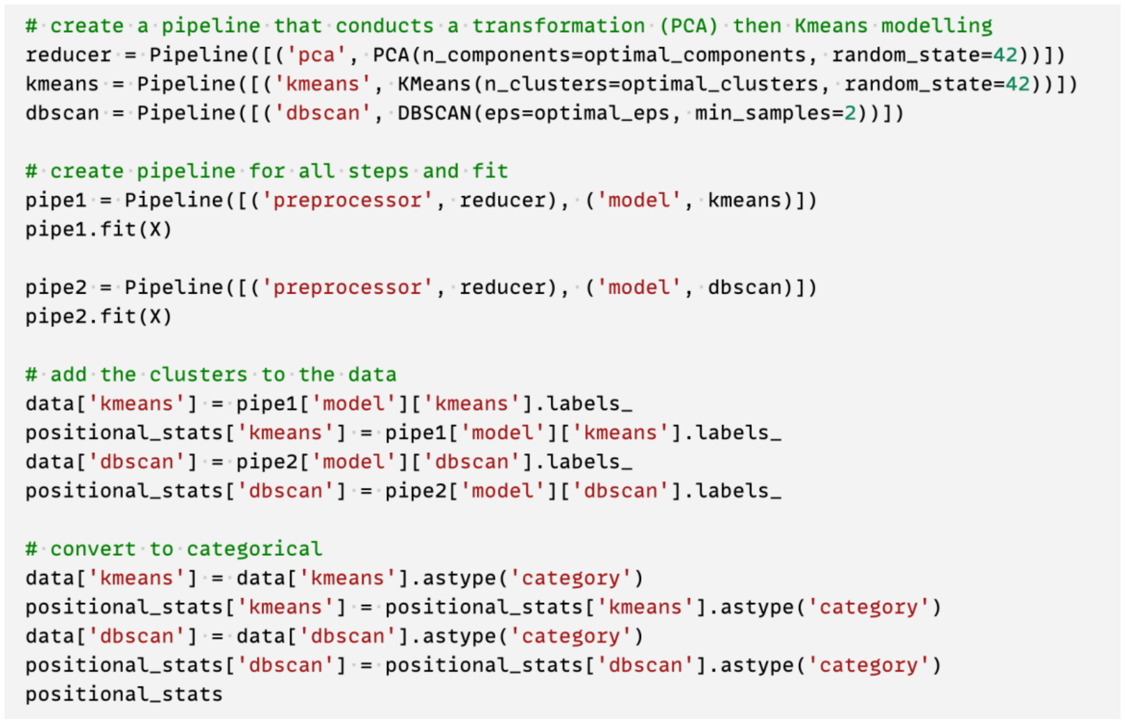


**Figure 12 –** K-distance (optimal eps) by observations.



```
# create a pipeline that conducts a transformation (PCA) then Kmeans modelling
reducer = Pipeline([('pca', PCA(n_components=optimal_components, random_state=42))])
kmeans = Pipeline([('kmeans', KMeans(n_clusters=optimal_clusters, random_state=42))])
dbscan = Pipeline([('dbscan', DBSCAN(eps=optimal_eps, min_samples=2))])

# create pipeline for all steps and fit
pipe1 = Pipeline([('preprocessor', reducer), ('model', kmeans)])
pipe1.fit(X)

pipe2 = Pipeline([('preprocessor', reducer), ('model', dbscan)])
pipe2.fit(X)

# add the clusters to the data
data['kmeans'] = pipe1['model']['kmeans'].labels_
positional_stats['kmeans'] = pipe1['model']['kmeans'].labels_
data['dbscan'] = pipe2['model']['dbscan'].labels_
positional_stats['dbscan'] = pipe2['model']['dbscan'].labels_

# convert to categorical
data['kmeans'] = data['kmeans'].astype('category')
positional_stats['kmeans'] = positional_stats['kmeans'].astype('category')
data['dbscan'] = data['dbscan'].astype('category')
positional_stats['dbscan'] = positional_stats['dbscan'].astype('category')
positional_stats
```

**Figure 13 –** Code for final modeling of KMeans and DBSCAN models using optimal parameters.

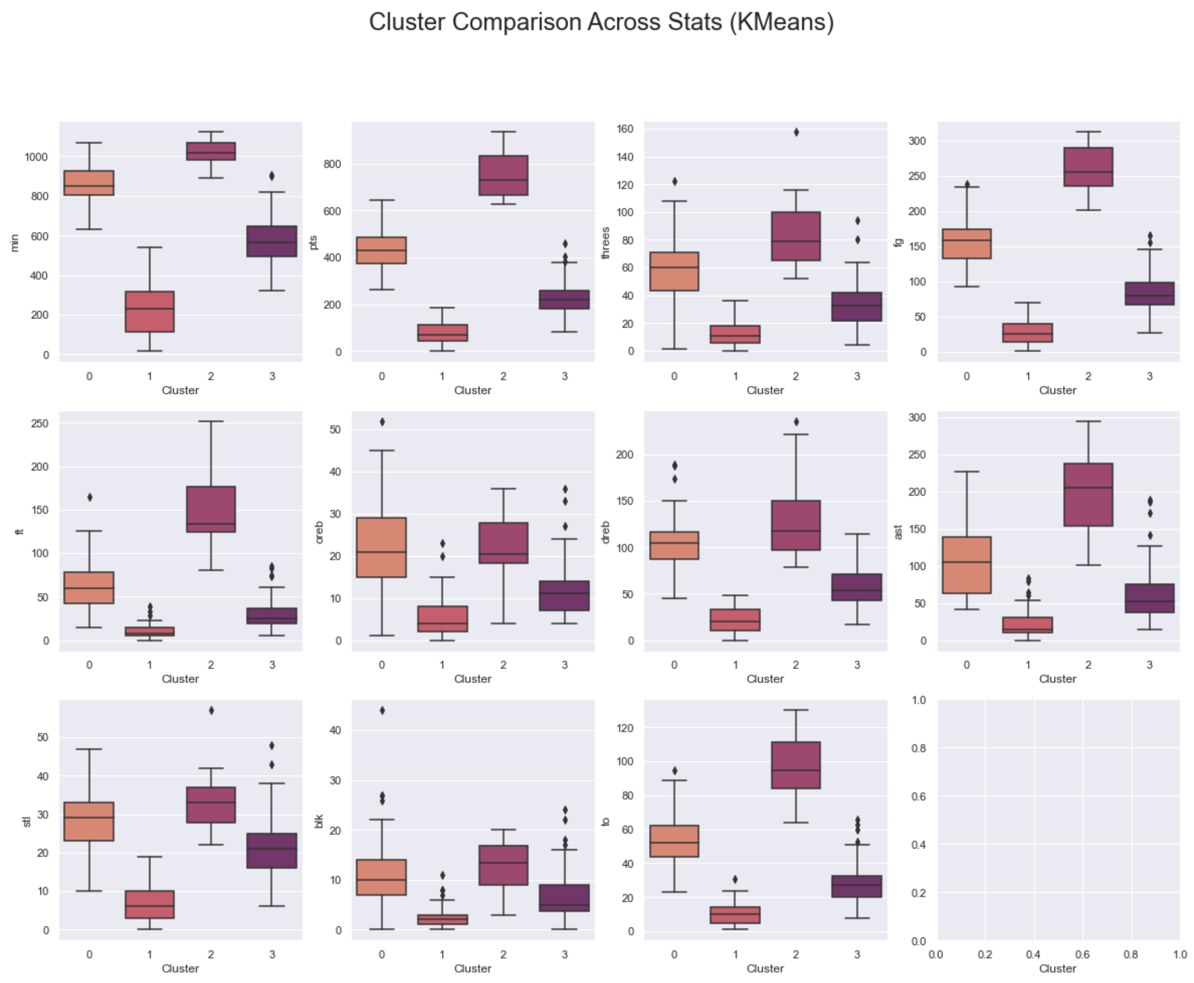


**Figure 14 –** Boxplots for player statistics by cluster assignment (KMeans)

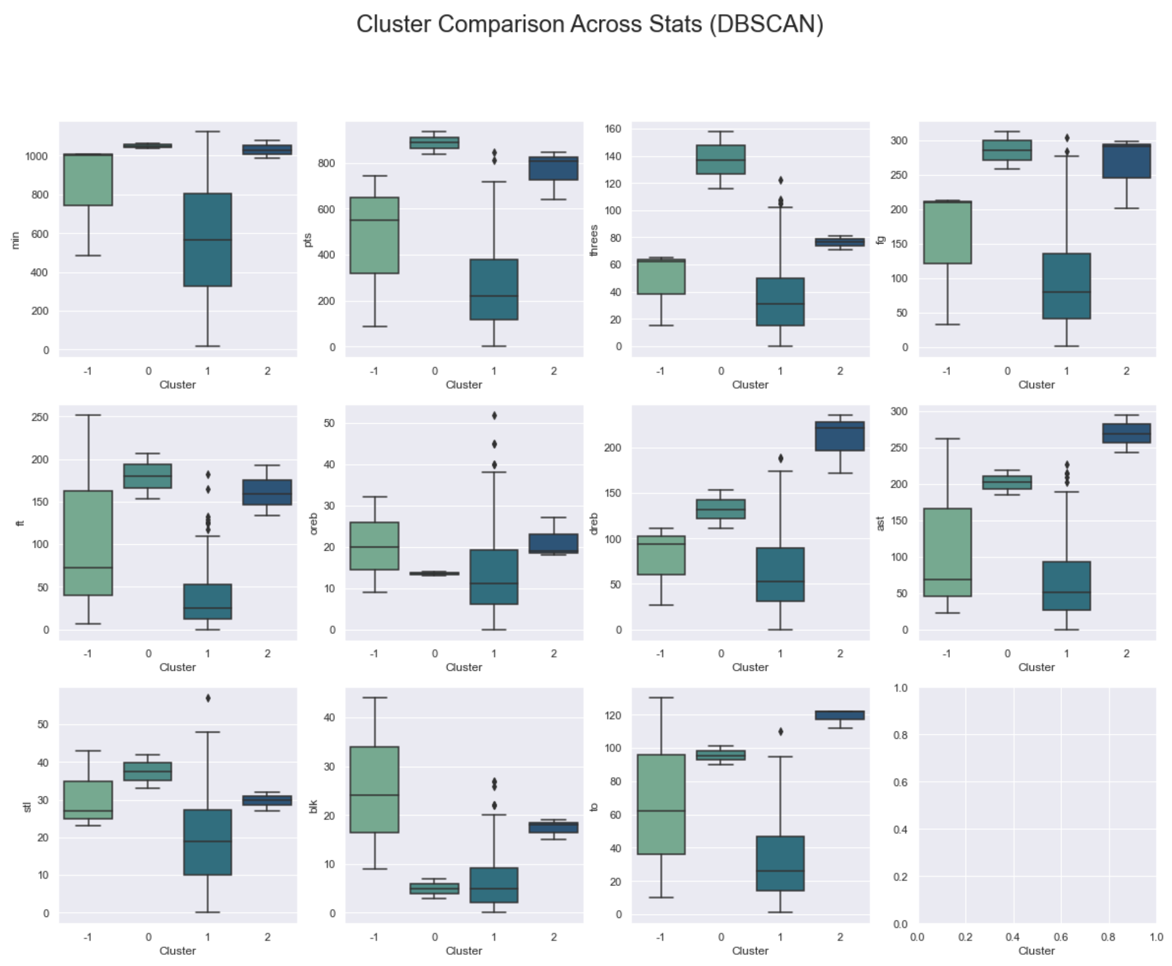
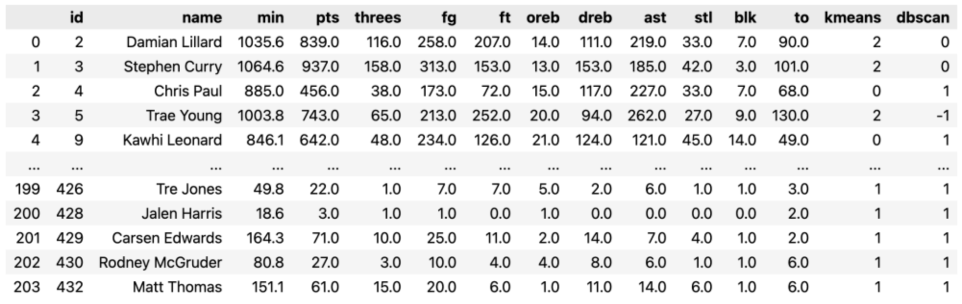


**Figure 15 –** Boxplots for player statistics by cluster assignment (DBSCAN)



| | id | name | min | pts | threes | fg | ft | oreb | dreb | ast | stl | blk | to | kmeans | dbscan |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | Damian Lillard | 1035.6 | 839.0 | 116.0 | 258.0 | 207.0 | 14.0 | 111.0 | 219.0 | 33.0 | 7.0 | 90.0 | 2 | 0 |
| 1 | 3 | Stephen Curry | 1064.6 | 937.0 | 158.0 | 313.0 | 153.0 | 13.0 | 153.0 | 185.0 | 42.0 | 3.0 | 101.0 | 2 | 0 |
| 2 | 4 | Chris Paul | 885.0 | 456.0 | 38.0 | 173.0 | 72.0 | 15.0 | 117.0 | 227.0 | 33.0 | 7.0 | 68.0 | 0 | 1 |
| 3 | 5 | Trae Young | 1003.8 | 743.0 | 65.0 | 213.0 | 252.0 | 20.0 | 94.0 | 262.0 | 27.0 | 9.0 | 130.0 | 2 | -1 |
| 4 | 9 | Kawhi Leonard | 846.1 | 642.0 | 48.0 | 234.0 | 126.0 | 21.0 | 124.0 | 121.0 | 45.0 | 14.0 | 49.0 | 0 | 1 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 199 | 426 | Tre Jones | 49.8 | 22.0 | 1.0 | 7.0 | 7.0 | 5.0 | 2.0 | 6.0 | 1.0 | 1.0 | 3.0 | 1 | 1 |
| 200 | 428 | Jalen Harris | 18.6 | 3.0 | 1.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 2.0 | 1 | 1 |
| 201 | 429 | Carsen Edwards | 164.3 | 71.0 | 10.0 | 25.0 | 11.0 | 2.0 | 14.0 | 7.0 | 4.0 | 1.0 | 2.0 | 1 | 1 |
| 202 | 430 | Rodney McGruder | 80.8 | 27.0 | 3.0 | 10.0 | 4.0 | 4.0 | 8.0 | 6.0 | 1.0 | 1.0 | 6.0 | 1 | 1 |
| 203 | 432 | Matt Thomas | 151.1 | 61.0 | 15.0 | 20.0 | 6.0 | 1.0 | 11.0 | 14.0 | 6.0 | 1.0 | 6.0 | 1 | 1 |

**Figure 16 –** Resulting dataframe (for Guard positions) with cluster assignments included.