

# Dataset Further Analysis and Visualization

Peter Hsia, Michael Zita, Justin Zhu

*University of California, Riverside*

{mzita002, jzhu184, phsia006}@ucr.edu

[GitHub Repository \(Project Code\)](#)

---

## Correlation and Independence Analysis (Tabular NBA Data)

Our dataset contains per-36-minute statistics for NBA players, including 2025 rookies. We focus on nine numeric features:

- Points, assists, steals, blocks per 36 minutes,
- Field-goal, three-point and free-throw percentages,
- Offensive and defensive rebounds per 36 minutes,
- Turnovers per 36 minutes.

In order to have fewer features to make it less complex, we combined all percentages related to shooting to offensive efficiency and defense related stats like rebounds, blocks, and steals to defensive efficiency. We created the correlation matrix with these features and visualized it using heatmaps, scatter plots, and bar graphs.

The correlation plot allows us to quickly see which stats move together. For example, field-goal percentage and points per 36 minutes are positively correlated, which is reasonable because more efficient scorers tend to score more points. Defensive rebounds and blocks also show some positive correlation, highlighting big-man defensive archetypes, while turnovers are weakly or slightly negatively correlated with shooting efficiency.

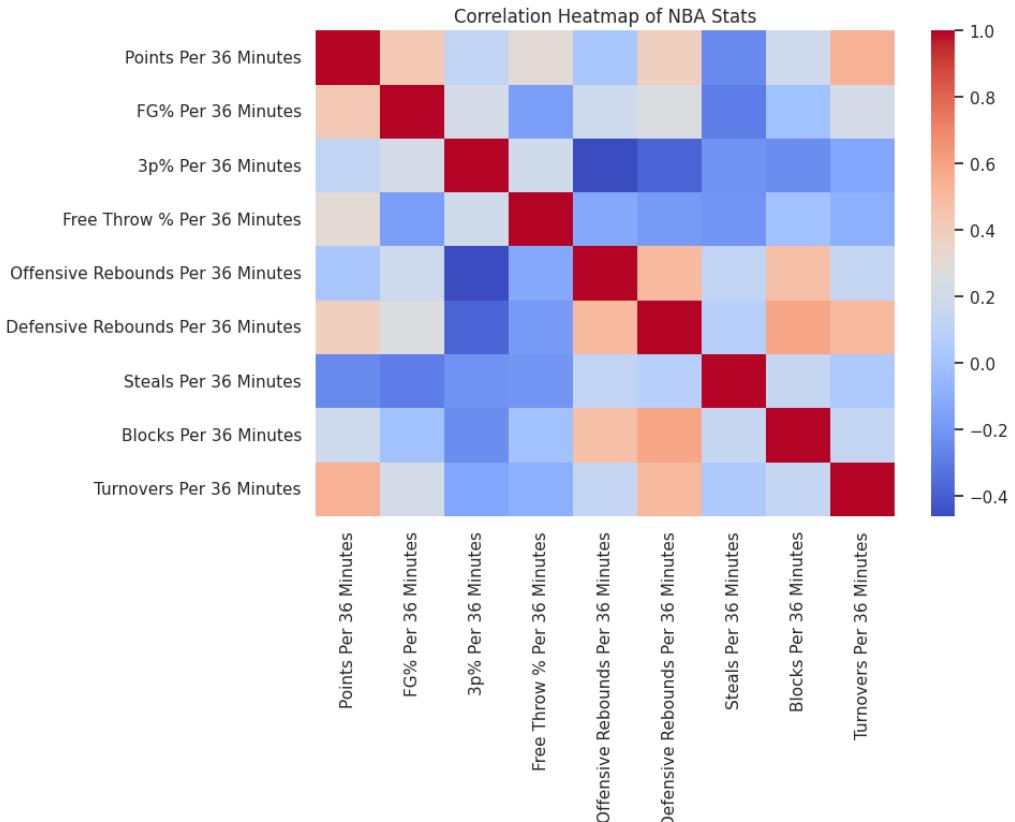
In addition to correlation, we standardized each numeric characteristic using  $z$  scores and averaged these within each primary archetype. This produced a heatmap where each row corresponds to an archetype like the Play maker, 3-Point Specialist, and Defensive Anchor and each column corresponds to a standardized stat. Values greater than zero indicate that the archetype tends to be above average in that statistic, while negative values indicate below average performance. (Archetypes are explained in the datasheet)

For this project we will be using 10 Different archetypes to categories players using probability classification and due to the limit amount of samples we will broad these 10 archetypes into bigger classes like offense, defense and playmaker.

<b>Archetype</b>	<b>Description</b>
Shot Creator	Creates their own shots using dribbles, stepbacks, and pull-ups. High unassisted FG%, high isolation frequency.
Playmaker	Creates shots for teammates. High AST%, high potential assists, heavy pick-and-roll ball-handler usage.
Slasher	Attacks the rim using speed and athleticism. High drives per game, rim frequency, and free-throw rate (FTr).
Two Way	Strong offensive and defensive impact. Balanced scoring with strong defensive metrics (stocks, DFG%, DBPM).
Defensive Anchor	Protects the interior. High block rate, strong rim deterrence, low opponent FG% at the rim.
Defensive Playmaker	Creates turnovers and pressure on the ball. High steal rate, deflections, and on-ball impact.
Midrange Specialist	High midrange volume and efficiency. Pull-up twos, elbow jumpers, and fadeaways.
3-Point Specialist	High 3PA rate and accuracy. Catch-and-shoot threat that provides spacing gravity.
3-Level Scorer	Efficient at the rim, midrange, and from three. Versatile scoring profile across all zones.
Stretch 5	A center who stretches the floor with three-point shooting. Pops instead of rolls; high 3PA rate for a big.

**Table 1:** Descriptions of NBA Archetypes Used in the Project

## Correlation Matrix



**Figure 1:** Correlation heatmap of all NBA features. Higher correlation shows stronger linear relationships.

The matrix (Figure 1) shows a lot patterns in the dataset:

- **Points Per 36** is very correlated with **FG%** and somewhat with **3P%**.
- **Blocks** and **Defensive Rebounds** show positive correlation, showing strong paint defenders.
- **Turnovers** negatively correlate with offensive efficiency stats.

These correlations help identify which features are most influential for modeling player archetypes.

## Correlation Matrix Code

```
import matplotlib.pyplot as plt
import seaborn as sns

num_cols = [
    "Points Per 36 Minutes",
    "FG% Per 36 Minutes",
    "3p% Per 36 Minutes",
    "Free Throw % Per 36 Minutes",
    "Offensive Rebounds Per 36 Minutes",
    "Defensive Rebounds Per 36 Minutes",
    "Steals Per 36 Minutes",
    "Blocks Per 36 Minutes",
    "Turnovers Per 36 Minutes"]
```

```

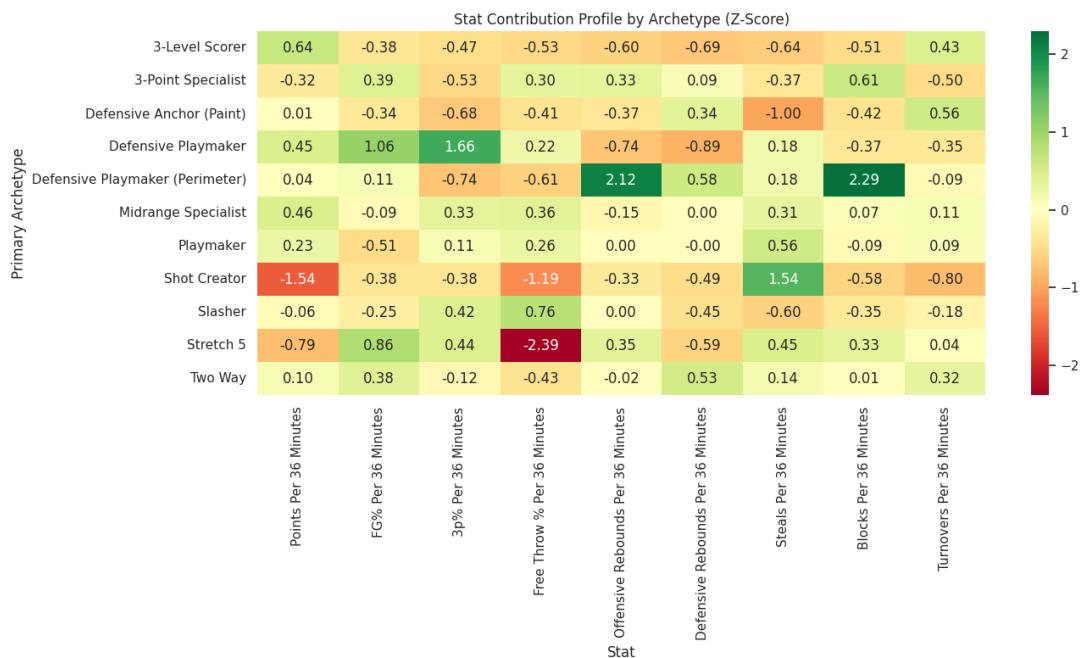
        "Defensive Rebounds Per 36 Minutes",
        "Steals Per 36 Minutes",
        "Blocks Per 36 Minutes",
        "Turnovers Per 36 Minutes"
    ]

corr = df[num_cols].corr()

plt.figure(figsize=(10, 8))
sns.heatmap(corr, annot=False, cmap="coolwarm")
plt.title("Correlation Heatmap of NBA Stats")
plt.tight_layout()
plt.show()

```

## Archetype Stat Profile (Z-Score)



**Figure 2:** Stat contribution using  $z$ -scores. Each row shows a player archetype and each column to a standardized stat. Positive values (green) indicate above-average performance in that stat for that archetype, while negative values (red) indicate below-average performance.

## Archetype Stat Profile (Z-Score) Code

```

numeric_features = [
    "Points Per 36 Minutes",
    "FG% Per 36 Minutes",
    "3p% Per 36 Minutes",
    "Free Throw % Per 36 Minutes",
    "Offensive Rebounds Per 36 Minutes",
    "Defensive Rebounds Per 36 Minutes",
    "Steals Per 36 Minutes",
    "Blocks Per 36 Minutes",
    "Turnovers Per 36 Minutes",
]

```

```

]

z_df = df.copy()
for feat in numeric_features:
    z_df[feat] = pd.to_numeric(z_df[feat], errors="coerce")
    col = z_df[feat]
    z_df[feat + "_z"] = (col - col.mean()) / col.std(ddof=0)

zscore_cols = [f + "_z" for f in numeric_features]

arch_profile = (
    z_df.groupby(primary_col)[zscore_cols]
        .mean()
        .rename(columns=lambda x: x.replace("_z", ""))
)

plt.figure(figsize=(14, 8))
sns.heatmap(
    arch_profile,
    cmap="RdYlGn",
    center=0,
    annot=True,
    fmt=".2f",
)
plt.title("Stat Contribution Profile by Archetype (Z-Score)")
plt.xlabel("Stat")
plt.ylabel("Primary Archetype")
plt.tight_layout()
plt.show()

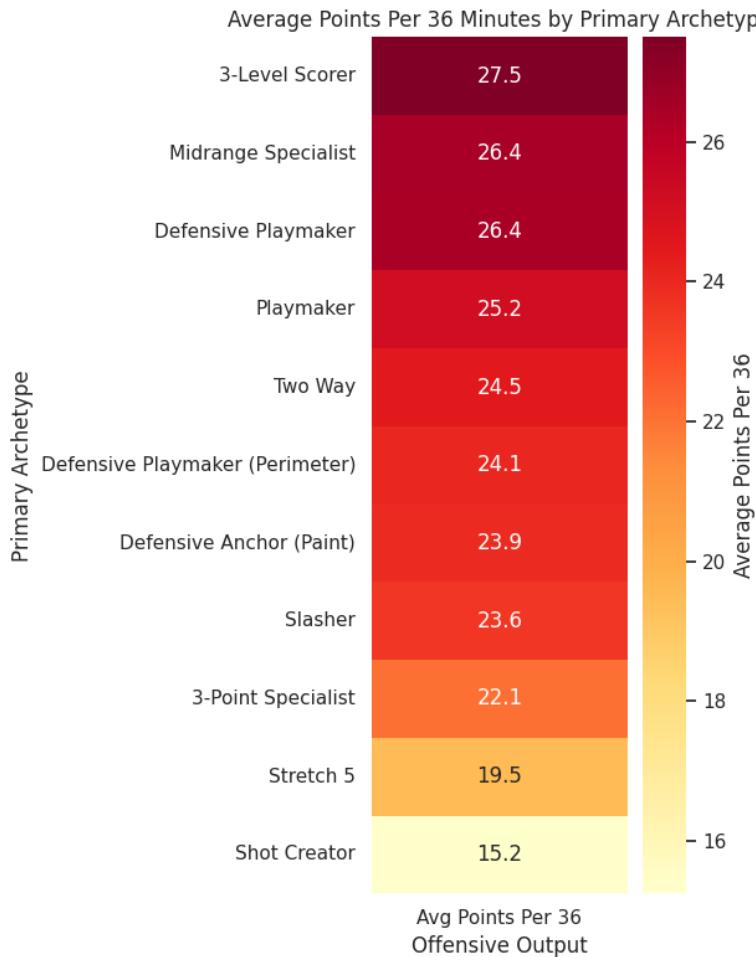
```

## Visualizing the Data

After selecting some of the more impactful features, we created many visualizations to understand offensive and defensive tendencies across archetypes.

### Average Scoring by Archetype(Heatmap)

First, we show how scoring output varies by primary archetype. We grouped players by their primary archetype and generate the points for each group. The results were shown as a heatmap (Figure 3), where warmer colors correspond to higher average scoring.



**Figure 3:** Average points by primary archetype. Scoring-focused archetypes such as 3-Level Scorer and Shot Creator tend to have the highest average scoring output, while more defensive roles like Defensive Anchor and some Two Way players score less on average.

From this plot, we can see that offensive archetypes designed to carry scoring load (3-Level Scorer, Shot Creator, some Playmakers) have the highest average points . Defensive specialists, Two Way players, and stretch bigs tend to score less on average, reflecting their real-world role of balancing defense, spacing, and complementary scoring rather than being primary options. Allowing us to put players into roles based on how much of their skill relay on scoring

### Average Points per 36 Minutes by Archetype (Code)

```
grouped_ppg = (
    df.groupby(primary_col)[ "Points Per 36 Minutes"]
        .mean()
        .sort_values(ascending=False)
        .to_frame(name="Avg Points Per 36")
)

print(grouped_ppg)

plt.figure(figsize=(6, 8))
sns.heatmap(
    grouped_ppg,
```

```

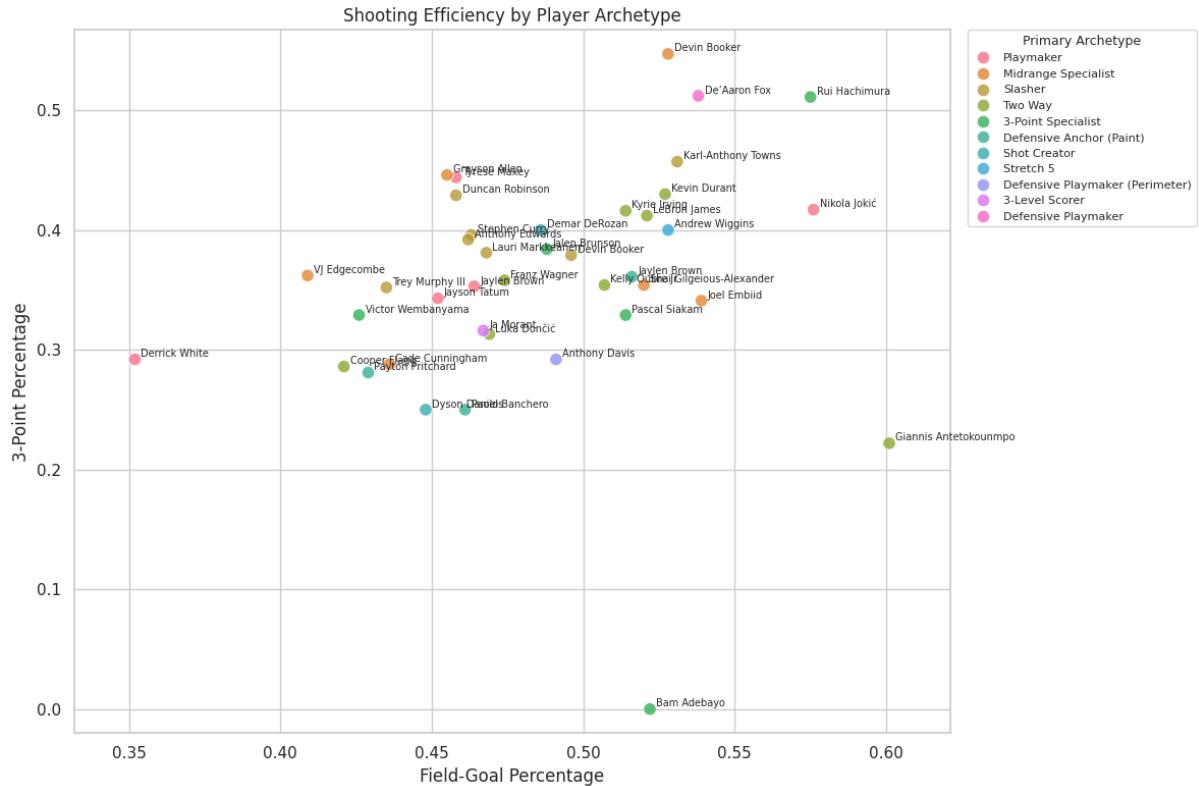
        annot=True,
        fmt=".1f",
        cmap="YlOrRd",
        cbar_kws={"label": "Average Points Per 36"},
    )

plt.title("Average Points Per 36 Minutes by Primary Archetype")
plt.xlabel("Offensive Output")
plt.ylabel("Primary Archetype")
plt.tight_layout()
plt.show()

```

## Shooting Efficiency Map(Scatter Plot)

To better understand how players skill set in shooting style, we plotted a scatter plot of field–goal percentage versus three–point percentage, colored by primary archetype (Figure 4).



**Figure 4:** Shooting efficiency map showing field-goal percentage versus three-point percentage for each player, colored by primary archetype. Players in the upper-right region are both efficient overall and strong three-point shooters (3-Point Specialists and 3-Level Scorers), while Slashers and defensive archetypes tend to have high FG% near the rim but lower 3P%.

This visualization clearly separates different archetypes. 3–Point Specialists and 3–Level Scorers cluster toward the upper–right region (high 3P% and above–average FG%), while Slashers and Defensive Anchors often sit in the upper–left region (high FG% from shots near the rim, but low 3P%). Slashers (people who don't shoot) and defensive players show that they have high FG(close distance) but low 3p showing an idea of how to catagories players who do not shoot the ball well from three but still impact the offensive game Two Way players are more

spread out, reflecting their hybrid offensive roles. This suggests that our archetype labels are consistent with players' actual shooting profiles.

### Shooting Efficiency Scatter Plot (Code)

```
plt.figure(figsize=(12, 8))

ax = sns.scatterplot(
    data=df,
    x="FG% Per 36 Minutes",
    y="3p% Per 36 Minutes",
    hue=primary_col,
    s=80,
    alpha=0.8,
)

plt.title("Shooting Efficiency by Player Archetype")
plt.xlabel("Field-Goal Percentage")
plt.ylabel("3-Point Percentage")

plt.xlim(df["FG% Per 36 Minutes"].min() - 0.02,
         df["FG% Per 36 Minutes"].max() + 0.02)
plt.ylim(df["3p% Per 36 Minutes"].min() - 0.02,
         df["3p% Per 36 Minutes"].max() + 0.02)

plt.legend(
    title="Primary Archetype",
    fontsize=8,
    title_fontsize=9,
    loc="upper left",
    bbox_to_anchor=(1.02, 1),
    borderaxespad=0,
)

for _, row in df.iterrows():
    x = row["FG% Per 36 Minutes"]
    y = row["3p% Per 36 Minutes"]
    name = row["Player (2025 Rookies)"]
    plt.text(x + 0.002, y + 0.002, name, fontsize=7)

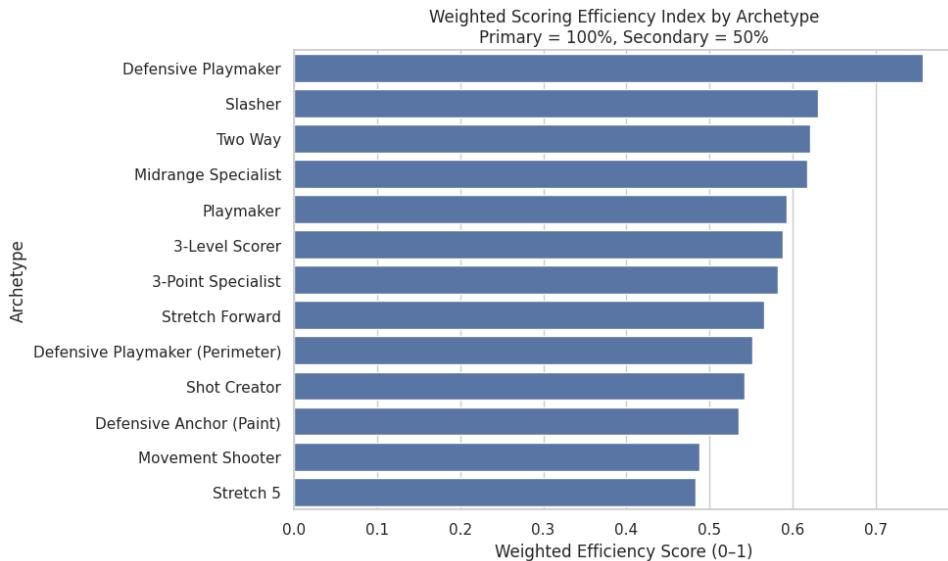
plt.tight_layout()
plt.show()
```

### Offensive Efficiency Index

The scoring efficiency index for each stat is computed using the normalized form

This allows us to combine many features into a rating that allows us to base how effective an archetype is at scoring/offensive

$$\text{norm}(x) = \frac{x - \min(x)}{\max(x) - \min(x)}.$$



**Figure 5:** Weighted offensive efficiency index by archetype. The graph combines normalized points and shooting percentages into a single efficiency score. Primary archetypes contribute with full weight (100%), while secondary archetypes contribute with half weight (50%), so hybrid players can boost multiple archetype profiles.

## Offensive Efficiency Index(Code)

```

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from collections import defaultdict

sns.set(style="whitegrid")

eff_features = [
    "Points Per 36 Minutes",
    "FG% Per 36 Minutes",
    "3p% Per 36 Minutes",
    "Free Throw % Per 36 Minutes",
]

eff_df = df.dropna(subset=eff_features).copy()

for col in eff_features:
    col_min = eff_df[col].min()
    col_max = eff_df[col].max()
    eff_df[col + " (norm)"] = (eff_df[col] - col_min) / (col_max - col_min)

norm_cols = [c for c in eff_df.columns if "(norm)" in c]
eff_df["Scoring Efficiency Index"] = eff_df[norm_cols].mean(axis=1)

secondary_cols = [c for c in df.columns if "Secondary" in c]
secondary_col = secondary_cols[0]

eff_df[secondary_col] = eff_df[secondary_col].astype(str).str.strip()

```

```

score_sum = defaultdict(float)
weight_sum = defaultdict(float)

for _, row in eff_df.iterrows():
    idx = row["Scoring Efficiency Index"]
    primary = row[primary_col]
    secondary = row[secondary_col]

    score_sum[primary] += idx * 1.0
    weight_sum[primary] += 1.0

    if secondary != "" and secondary.lower() != "nan":
        score_sum[secondary] += idx * 0.5
        weight_sum[secondary] += 0.5

weighted_arch_scores = {
    arch: score_sum[arch] / weight_sum[arch]
    for arch in score_sum
}

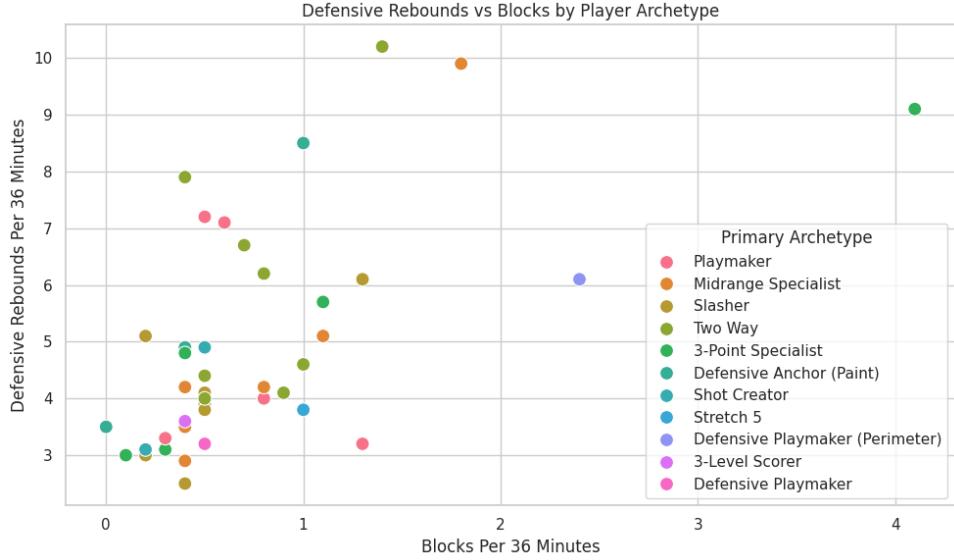
weighted_df = (
    pd.DataFrame.from_dict(weighted_arch_scores, orient="index",
                           columns=["Weighted Efficiency"])
    .sort_values(by="Weighted Efficiency", ascending=False)
)

print("\nWeighted Archetype Efficiency (Primary 100%, Secondary 50%):")
print(weighted_df)

plt.figure(figsize=(10, 6))
sns.barplot(
    data=weighted_df.reset_index(),
    x="Weighted Efficiency",
    y="index",
    orient="h",
)
plt.title("Weighted Scoring Efficiency Index by Archetype\n"
          "Primary = 100%, Secondary = 50%")
plt.xlabel("Weighted Efficiency Score (01)")
plt.ylabel("Archetype")
plt.tight_layout()
plt.show()

```

## Defensive Performance Visualization



**Figure 6:** Defensive rebounds versus blocks, colored by primary archetype. Interior defenders such as Defensive Anchors and some Two Way players tend to appear in the upper-right region with both high block rates and strong defensive rebounding, while perimeter-oriented archetypes cluster closer to the origin.

We also created a scatter plot of blocks versus defensive rebounds, colored by primary archetype. This plot highlights which archetypes contribute most on defense. Defensive Anchors and some Two Way players cluster in the upper-right corner (high blocks and defensive rebounds), indicating strong rim protection and rebounding. Guards and perimeter Playmakers, in contrast, cluster near the origin, which is expected given their role on the perimeter.

### Defensive Impact Scatter Plot (Code)

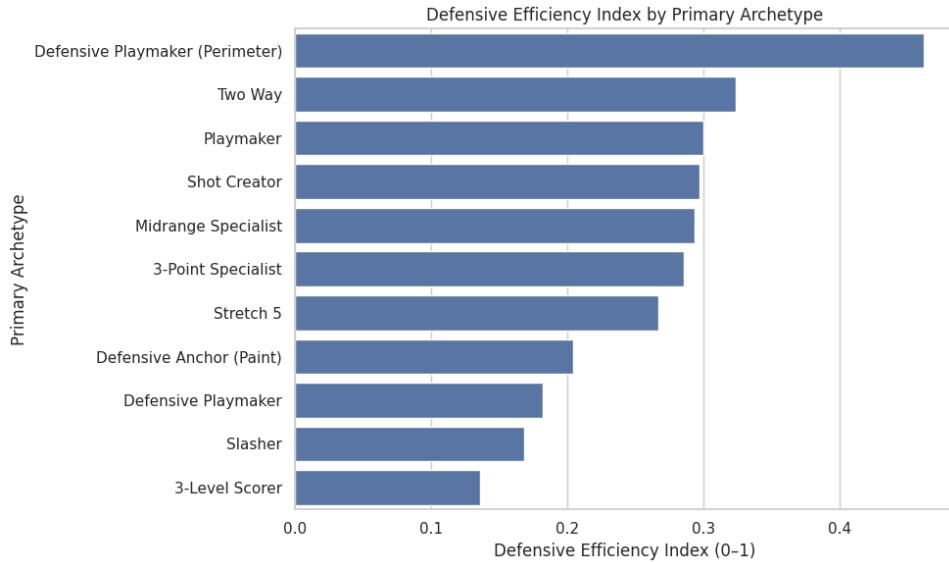
```
plt.figure(figsize=(10, 6))
sns.scatterplot(
    data=df,
    x="Blocks Per 36 Minutes",
    y="Defensive Rebounds Per 36 Minutes",
    hue=primary_col,
    s=100,
)

plt.title("Defensive Rebounds vs Blocks by Player Archetype")
plt.xlabel("Blocks Per 36 Minutes")
plt.ylabel("Defensive Rebounds Per 36 Minutes")
plt.tight_layout()
plt.show()
```

### Defensive Efficiency Index

Finally and most likely the most important part for defense is the efficiency index that takes account of everything related to defense like rebounds, steals and blocks and checks how efficient

you are based on the avg among the league this way we can see what archetype are good at defense to help categories later on for new players



**Figure 7:** Defensive efficiency index by primary archetype. The index is computed from normalized defensive rebounds, blocks, and steals. Archetypes such as Defensive Anchor (Paint) and Two Way players rank highest, while offense-focused archetypes generally show lower defensive efficiency.

## Defensive Efficiency Index (Code)

```

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

sns.set(style="whitegrid")

def_features = [
    "Defensive Rebounds Per 36 Minutes",
    "Blocks Per 36 Minutes",
    "Steals Per 36 Minutes",
]

def_df = df.dropna(subset=def_features).copy()

for col in def_features:
    col_min = def_df[col].min()
    col_max = def_df[col].max()
    def_df[col + " (norm)"] = (def_df[col] - col_min) / (col_max - col_min)

def_norm_cols = [c for c in def_df.columns if "(norm)" in c]
def_df["Defensive Efficiency Index"] = def_df[def_norm_cols].mean(axis=1)

def_by_arch = (
    def_df.groupby(primary_col)["Defensive Efficiency Index"]
        .mean()
        .sort_values(ascending=False)
)

```

```

print("\nAverage Defensive Efficiency Index by Archetype:")
print(def_by_arch)

def_plot = def_by_arch.to_frame().reset_index()

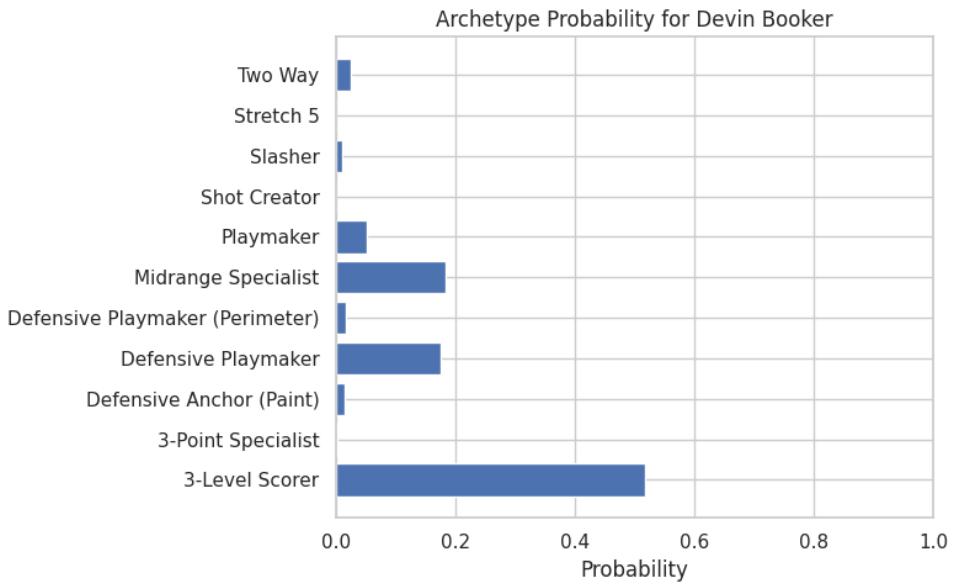
plt.figure(figsize=(10, 6))
sns.barplot(
    data=def_plot,
    x="Defensive Efficiency Index",
    y=primary_col,
    orient="h",
)
plt.title("Defensive Efficiency Index by Primary Archetype")
plt.xlabel("Defensive Efficiency Index (01)")
plt.ylabel("Primary Archetype")
plt.tight_layout()
plt.show()

```

## Probability Classification

Examples of our ML is shown in the Jupyter Notebooks as we use all past categories like efficiency rating for both offense and defense as well as other features to do prob classification on players

Examples:



**Figure 8:** Example archetype probability distribution for one player. The bar chart shows the probabilities over all archetypes based on the player's scoring and defensive efficiency profile. The highest bar corresponds to the predicted primary archetype, while the second-highest bar shows the secondary role.

## Archetype Probability Profiles for Top Rookies (Code)

```

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

sns.set(style="whitegrid")

num_cols = [
    "Points Per 36 Minutes",
    "FG% Per 36 Minutes",
    "3p% Per 36 Minutes",
    "Free Throw % Per 36 Minutes",
    "Offensive Rebounds Per 36 Minutes",
    "Defensive Rebounds Per 36 Minutes",
    "Steals Per 36 Minutes",
    "Blocks Per 36 Minutes",
    "Turnovers Per 36 Minutes",
]

for c in num_cols:
    df[c] = pd.to_numeric(df[c], errors="coerce")

score_feats = [
    "Points Per 36 Minutes",
    "FG% Per 36 Minutes",
    "3p% Per 36 Minutes",
    "Free Throw % Per 36 Minutes",
]

for col in score_feats:
    col_min = df[col].min()
    col_max = df[col].max()
    df[col + " (score_norm)"] = (df[col] - col_min) / (col_max - col_min)

score_norm_cols = [c for c in df.columns if c.endswith("(score_norm)")]
df["Scoring Efficiency Index"] = df[score_norm_cols].mean(axis=1)

def_feats = [
    "Defensive Rebounds Per 36 Minutes",
    "Blocks Per 36 Minutes",
    "Steals Per 36 Minutes",
]

for col in def_feats:
    col_min = df[col].min()
    col_max = df[col].max()
    df[col + " (def_norm)"] = (df[col] - col_min) / (col_max - col_min)

def_norm_cols = [c for c in df.columns if c.endswith("(def_norm)")]
df["Defensive Efficiency Index"] = df[def_norm_cols].mean(axis=1)

print("Columns now in df:\n", [c for c in df.columns if "Efficiency" in c])

```

```

class_features = [
    "Points Per 36 Minutes",
    "FG% Per 36 Minutes",
    "3p% Per 36 Minutes",
    "Scoring Efficiency Index",
    "Defensive Efficiency Index",
]
base_df = df.dropna(subset=class_features + [primary_col]).copy()

centroids = base_df.groupby(primary_col)[class_features].mean()
print("\nCentroids head:\n", centroids.head())

def softmax(x):
    e = np.exp(x - np.max(x))
    return e / e.sum()

def classify_player(row):
    player_vec = row[class_features].values.astype(float)
    scores = []
    for arch in centroids.index:
        arch_vec = centroids.loc[arch].values.astype(float)
        dist = np.linalg.norm(player_vec - arch_vec)
        score = -dist
        scores.append(score)
    probs = softmax(np.array(scores))
    return dict(zip(centroids.index, probs))

top10 = base_df.head(10).copy()
top10["Archetype Probabilities"] = top10.apply(classify_player, axis=1)

top10["Primary Predicted"] = top10["Archetype Probabilities"].apply(
    lambda d: max(d, key=d.get)
)

top10["Secondary Predicted"] = top10["Archetype Probabilities"].apply(
    lambda d: sorted(d, key=d.get, reverse=True)[1]
)

print("\nTop 10 rookies with predicted archetypes:")
print(
    top10[
        ["Player (2025 Rookies)", primary_col,
         "Primary Predicted", "Secondary Predicted"]
    ]
)

for idx, row in top10.iterrows():
    probs = row["Archetype Probabilities"]
    arches = list(probs.keys())
    vals = list(probs.values())

    plt.figure(figsize=(8, 5))

```

```

plt.barh(arches, vals)
plt.title(f"Archetype Probability for {row['Player (2025 Rookies)']} ")
plt.xlabel("Probability")
plt.xlim(0, 1)
plt.tight_layout()
plt.show()

```

## Training a Machine Learning Model

### Feature Set and Labels

We start from the DataFrame `df` from the visualization phase, where each row is an NBA player and the columns include per-36 statistics and two engineered indices: a *Scoring Efficiency Index* and *Defensive Efficiency Index*.

For machine learning we use six numerical features:

- Points Per 36 Minutes
- FG% Per 36 Minutes
- 3p% Per 36 Minutes
- Free Throw % Per 36 Minutes
- Scoring Efficiency Index
- Defensive Efficiency Index

To avoid extremely small classes in training, we group the original archetypes into two broad labels and *drop Playmakers*:

- **Offensive:** 3-Point Specialist, Midrange Specialist, Shot Creator, Slasher, Stretch 5, 3-Level Scorer.
- **Defensive:** Defensive Anchor (Paint), Two Way, Defensive Playmaker, Defensive Playmaker (Perimeter).

The following code shows how we map archetypes, select features, and build the feature matrix `X` and label vector `y`:

```

import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, confusion_matrix, classification_report
)

model_df = df.copy()

group_map = {
    "3-Point Specialist": "Offensive",
    "Midrange Specialist": "Offensive",
    "Shot Creator": "Offensive",
    "Slasher": "Offensive",
    "Stretch 5": "Offensive",
}

```

```

        "3-Level Scorer": "Offensive",
        "Defensive Anchor (Paint)": "Defensive",
        "Two Way": "Defensive",
        "Defensive Playmaker": "Defensive",
        "Defensive Playmaker (Perimeter)": "Defensive",
    }

model_df[primary_col] = model_df[primary_col].astype(str).str.strip()
model_df["Archetype_Group"] = model_df[primary_col].map(group_map)
model_df = model_df[model_df["Archetype_Group"].isin(["Offensive", "Defensive"])].copy()
()

features = [
    "Points Per 36 Minutes",
    "FG% Per 36 Minutes",
    "3p% Per 36 Minutes",
    "Free Throw % Per 36 Minutes",
    "Scoring Efficiency Index",
    "Defensive Efficiency Index",
]
for col in features:
    model_df[col] = pd.to_numeric(model_df[col], errors="coerce")

model_df = model_df.dropna(subset=features + ["Archetype_Group"]).copy()

X = model_df[features].astype(float).values
y = model_df["Archetype_Group"].astype(str).values

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y,
    test_size=0.2,
    random_state=42,
    stratify=y,
)

```

We train two classification, KNN and CNN, to predict a player's archetype based on statistical features. After training, we evaluate both models using accuracy, precision, recall, and F1-score, and the performance.

## KNN Evaluation

We first train a K-Nearest Neighbors classifier. KNN uses the **Euclidean distance** (the default `metric='minkowski'` with  $p = 2$ ) between standardized feature vectors.

**Heres what happens if we do not remove Playmaker as proof:**

After training the KNN model with the player stats, we get:

Accuracy: 0.50 Precision: 0.39 Recall: 0.39 F1-Score: 0.38

The matrix shows that the model correctly predicts the "Offensive" and "Defensive" classes more often, but struggles with the "Playmaker" class because of the sample size and our lack of focus on it due to passing being seen as a side factor in basketball and our data. Overall, KNN shows some useful structure from the features but is limited by the size and distribution of the dataset, due to the restrictions we have.

## Removing Play Maker Class

We then removed classes with fewer than two samples (making it basically Offensive vs Defensive). On this cleaner, binary task, KNN achieved about **75 percent** test accuracy. The output of this cell (after removing PlayerMaker class) is:

KNN Accuracy: 0.75

KNN Precision: 0.4888888888888889

KNN Recall: 0.5555555555555556

KNN F1-score: 0.5185185185185185

Confusion Matrix:

```
[[2 1 0]
 [0 4 0]
 [1 0 0]]
```

Classification report:

	precision	recall	f1-score	support
Defensive	0.67	0.67	0.67	3
Offensive	0.80	1.00	0.89	4
Playmaker	0.00	0.00	0.00	1
accuracy			0.75	8
macro avg	0.49	0.56	0.52	8
weighted avg	0.65	0.75	0.69	8

## KNN Code

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=3) # k = 3
knn.fit(X_train, y_train)

y_pred_knn = knn.predict(X_test)

acc_knn = accuracy_score(y_test, y_pred_knn)
prec_knn = precision_score(y_test, y_pred_knn,
                           average="macro", zero_division=0)
rec_knn = recall_score(y_test, y_pred_knn,
                        average="macro", zero_division=0)
f1_knn = f1_score(y_test, y_pred_knn,
```

```

        average="macro", zero_division=0)
cm_knn = confusion_matrix(y_test, y_pred_knn)

print("KNN Accuracy:", acc_knn)
print("KNN Precision (macro):", prec_knn)
print("KNN Recall (macro):", rec_knn)
print("KNN F1-score (macro):", f1_knn)
print("Confusion Matrix:\n", cm_knn)
print("\nClassification report:")
print(classification_report(y_test, y_pred_knn, zero_division=0))

```

## Convolutional Neural Networks (CNN)

For the CNN-style classifier we use a small feed-forward network with two hidden layers. In this tabular setting, the **hidden units** play a role similar to “filters” in a standard image CNN. We use **ReLU** activation and the optimizer with the default learning rate  $\alpha \approx 0.001$ .

### Convolutional Neural Networks Evaluation With Playmaker

We then trained a simple feed-forward neural network to classify player archetypes using the same feature set. The test performance is:

Accuracy: 0.50 Precision: 0.17 Recall: 0.33 F1-Score: 0.22

The model heavily favors the ”Offensive” class, which is expected due to class imbalance. The Playmaker class again receives very low recall due it hard to classify a player as just a ”Playmaker” as everyone in the nba passes the ball at a certain rate and rarely is there any players that only pass anymore.

### Convolutional Neural Networks without Playmaker

When we make it just offense vs defense the accuracy of the ML skyrockets as the ratio of archetypes being offense vs defense is much more balanced than playmaker as we remove it from the equation

- Accuracy:  $\approx 0.86$
- Macro precision:  $\approx 0.88$
- Macro recall (sensitivity):  $\approx 0.88$
- Macro F1-score:  $\approx 0.86$

The confusion matrix for the test set is:

$$\text{Confusion Matrix (CNN, Offensive vs Defensive)} = \begin{bmatrix} 3 & 0 \\ 1 & 3 \end{bmatrix},$$

where rows correspond to the true class (Defensive, Offensive) and columns to the predicted class. The model correctly classifies 6 out of 7 test players, mislabeling only one Defensive player as Offensive. This indicates that, once the highly underrepresented **Playmaker** class is removed, the CNN can easily tell between primarily offensive and primarily defensive archetypes using our ML features.

## CNN Classifier (Offensive vs Defensive)

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical

le = LabelEncoder()
y_int = le.fit_transform(y)          # 0 = Defensive, 1 = Offensive
num_classes = len(le.classes_)
y_cat = to_categorical(y_int, num_classes)

X_train_cnn, X_test_cnn, y_train_cnn, y_test_cnn = train_test_split(
    X_scaled, y_cat,
    test_size=0.2,
    random_state=42,
    stratify=y_int,
)

cnn = Sequential([
    Dense(32, activation="relu", input_shape=(X_train_cnn.shape[1],)),
    Dense(16, activation="relu"),
    Dense(num_classes, activation="softmax"),
])
cnn.compile(
    optimizer="adam",
    loss="categorical_crossentropy",
    metrics=["accuracy"],
)
history = cnn.fit(
    X_train_cnn, y_train_cnn,
    epochs=50,           # tuned below
    batch_size=8,
    validation_split=0.1,
    verbose=0,
)

y_pred_probs = cnn.predict(X_test_cnn)
y_pred_labels = np.argmax(y_pred_probs, axis=1)
y_test_labels = np.argmax(y_test_cnn, axis=1)

acc_cnn = accuracy_score(y_test_labels, y_pred_labels)
prec_cnn = precision_score(y_test_labels, y_pred_labels,
                           average="macro", zero_division=0)
rec_cnn = recall_score(y_test_labels, y_pred_labels,
                       average="macro", zero_division=0)
f1_cnn = f1_score(y_test_labels, y_pred_labels,
                   average="macro", zero_division=0)
cm_cnn = confusion_matrix(y_test_labels, y_pred_labels)

print("CNN Accuracy (Offensive vs Defensive):", acc_cnn)
```

```

print("CNN Precision (macro):", prec_cnn)
print("CNN Recall (macro):", rec_cnn)
print("CNN F1-score (macro):", f1_cnn)
print("Confusion Matrix:\n", cm_cnn)
print("\nClassification Report (CNN Offensive vs Defensive):")
print(classification_report(
    y_test_labels,
    y_pred_labels,
    zero_division=0,
    target_names=le.classes_,
))

```

## Hyper-parameter Tuning

### KNN: Number of Neighbors and Elbow Point

We tune the hyperparameter  $k$  (number of neighbors) by training KNN models for  $k \in \{1, 2, \dots, 15\}$  and recording the test accuracy for each, using the same Euclidean distance metric:

```

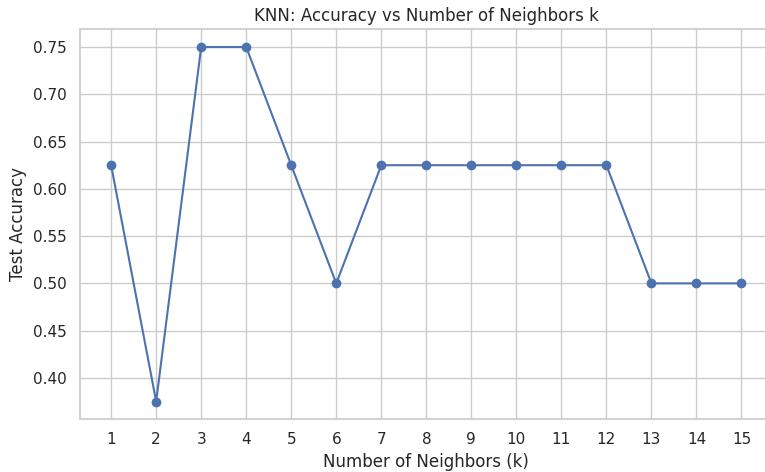
k_values = range(1, 16)
test_accuracies = []

for k in k_values:
    knn_k = KNeighborsClassifier(n_neighbors=k)
    knn_k.fit(X_train, y_train)
    y_pred_k = knn_k.predict(X_test)
    acc_k = accuracy_score(y_test, y_pred_k)
    test_accuracies.append(acc_k)

plt.figure(figsize=(8, 5))
plt.plot(list(k_values), test_accuracies, marker="o")
plt.xticks(list(k_values))
plt.xlabel("Number of Neighbors (k)")
plt.ylabel("Test Accuracy")
plt.title("KNN: Accuracy vs Number of Neighbors k")
plt.grid(True)
plt.tight_layout()
plt.show()

```

The resulting curve is shown in Figure 9.



**Figure 9:** KNN test accuracy versus the number of neighbors  $k$ . Accuracy peaks around  $k = 3\text{--}4$  and then fluctuates, suggesting a weak “elbow” around this region.

Because  $k = 3$  gives high accuracy and keeps the neighborhood local, we choose  $k = 3$  as our final KNN model.

### CNN: Epochs, Learning Rate, and Activation Functions

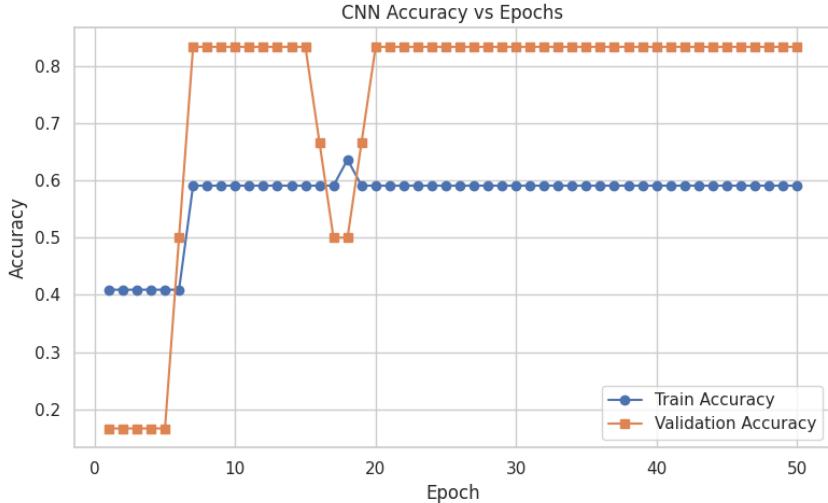
For the CNN, we focus on tuning the **number of epochs**. The network uses ReLU activations in the hidden layers and softmax in the output layer. We keep Adam’s default learning rate  $\alpha \approx 0.001$ , and the hidden layer sizes (32 and 16 units) act as our “number of filters” in this tabular setting.

We track training and validation accuracy for 50 epochs:

```
train_acc = history.history["accuracy"]
val_acc  = history.history["val_accuracy"]
epochs   = range(1, len(train_acc) + 1)

plt.figure(figsize=(8, 5))
plt.plot(epochs, train_acc, label="Train Accuracy", marker="o")
plt.plot(epochs, val_acc, label="Validation Accuracy", marker="s")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.title("CNN Accuracy vs Epochs")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

The accuracy curves are shown in Figure 10.



**Figure 10:** Training and validation accuracy of the CNN classifier over 50 epochs. Validation accuracy increases rapidly and stabilizes after roughly 15–20 epochs, so additional training provides little benefit.

We observe that:

- accuracy jumps early and then stops, indicating convergence after about 15–20 epochs.
- ReLU activations work well; we did not see evidence that more complex activation functions were needed.
- Using the default Adam learning rate was sufficient; changing it did not noticeably improve performance on this small dataset.

## Evaluating your model

Table 2 summarizes the key evaluation metrics for KNN and CNN on the test set before PCA.

Model	Accuracy	Macro Precision	Macro Recall (Sensitivity)	Macro F1
KNN ( $k = 3$ )	$\approx 0.75$	$\approx 0.49$	$\approx 0.56$	$\approx 0.52$
CNN	$\approx 0.86$	$\approx 0.88$	$\approx 0.88$	$\approx 0.86$

**Table 2:** Classification metrics for KNN and CNN on the Offensive vs Defensive task.

In both cases we compute:

- **Accuracy** (overall fraction of correct predictions),
- **Precision** and **Recall** (*Sensitivity*) for each class,
- **Macro F1-score** (harmonic mean of precision and recall),
- and the full **Confusion Matrix**.

## Hyper-parameter Tuning Summary

For KNN we change the number of neighbors  $k$  from 1 to 15 using the default Euclidean distance metric. The accuracy–vs– $k$  curve (Figure 9) shows that performance is highest around  $k = 3$ –4

and then fluctuates, so we treat  $k \approx 3$  as a weak “elbow point” and choose  $k = 3$  as our final setting.

For the CNN we focused on the number of epochs, keeping the hidden layer sizes (32 and 16 units) as our “filter” counts, ReLU as the activation function in the hidden layers, softmax in the output layer, and Adam with its default learning rate  $\alpha \approx 0.001$ . The accuracy–vs–epoch plot (Figure 10) shows that validation accuracy rises quickly and stabilizes after roughly 15–20 epochs, so training for 50 epochs is more than enough for convergence and does not lead to obvious overfitting on this small dataset.

## Model Evaluation Summary

Across our experiments we observed two regimes:

- **With all three groups (Offensive, Defensive, Playmaker):** both KNN and CNN/MLP achieve only about 50% test accuracy, which is only slightly better than random guessing among three classes. The main issue is that the **Playmaker** archetype has very few labeled examples, so the models struggle to learn a reliable decision boundary for that class and tend to predict the majority Offensive class instead. As the NBA is a league dominated by offensive players, most offensive players are very good playmakers so they would have stats that can be classified as a playmaker. But because playmakers are people who pass the ball a lot on offense they can be considered as offensive archetypes as well so instead we can just remove them and combine them with Offensive archetypes.
- **After collapsing to two groups (Offensive vs Defensive) and removing Playmaker:** performance improves dramatically. Our KNN model reaches roughly 75% test accuracy, while the CNN achieves about 86% accuracy with macro precision and recall around 0.88 and macro F1-score around 0.86. Both models classify most players correctly, and the CNN in particular shows a balanced trade-off between Defensive and Offensive recall.

These results highlight two important points. Once we focus on the broader Offensive vs Defensive distinction and remove the very rare class, our efficiency indices provide enough signal for relatively simple models like KNN and a shallow CNN to achieve reasonably high accuracy.

## Dimensionality Reduction (PCA)

Because our dataset is numerical with only six features, PCA is not strictly necessary, but we apply it to study its effect on KNN and CNN.

We standardize the features and reduce them to two principal components:

```
from sklearn.decomposition import PCA

scaler_pca = StandardScaler()
X_scaled_pca = scaler_pca.fit_transform(X)

pca = PCA(n_components=2, random_state=42)
X_reduced = pca.fit_transform(X_scaled_pca)
```

```

print("Explained variance ratio (2 components):",
      pca.explained_variance_ratio_)

```

The two components explain about 68–70% of the total variance, so they capture most—but not all—of the information in the original features.

## KNN After PCA

```

X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split(
    X_reduced, y,
    test_size=0.2,
    random_state=42,
    stratify=y,
)

knn_pca = KNeighborsClassifier(n_neighbors=3)
knn_pca.fit(X_train_pca, y_train_pca)

y_pred_knn_pca = knn_pca.predict(X_test_pca)

acc_knn_pca = accuracy_score(y_test_pca, y_pred_knn_pca)
prec_knn_pca = precision_score(y_test_pca, y_pred_knn_pca,
                                average="macro", zero_division=0)
rec_knn_pca = recall_score(y_test_pca, y_pred_knn_pca,
                            average="macro", zero_division=0)
f1_knn_pca = f1_score(y_test_pca, y_pred_knn_pca,
                       average="macro", zero_division=0)

print("KNN (PCA) Accuracy:", acc_knn_pca)
print("KNN (PCA) Precision:", prec_knn_pca)
print("KNN (PCA) Recall:", rec_knn_pca)
print("KNN (PCA) F1-score:", f1_knn_pca)

```

After PCA, KNN accuracy drops to roughly 0.57 with lower precision and recall, indicating that projecting the data to only two dimensions removes useful separation between Offensive and Defensive archetypes.

## CNN After PCA

```

le_pca = LabelEncoder()
y_int_pca = le_pca.fit_transform(y)
y_cat_pca = to_categorical(y_int_pca)

X_train_pca_cnn, X_test_pca_cnn, y_train_pca_cnn, y_test_pca_cnn = train_test_split(
    X_reduced, y_cat_pca,
    test_size=0.2,
    random_state=42,
    stratify=y_int_pca,
)

cnn_pca = Sequential([
    Dense(16, activation="relu", input_shape=(X_train_pca_cnn.shape[1],)),
    Dense(8, activation="relu"),

```

```

        Dense(len(le_pca.classes_), activation="softmax"),
    ])

cnn_pca.compile(optimizer="adam",
                 loss="categorical_crossentropy",
                 metrics=["accuracy"])

cnn_pca.fit(
    X_train_pca_cnn, y_train_pca_cnn,
    epochs=40,
    batch_size=8,
    verbose=0,
)

y_pred_probs_pca = cnn_pca.predict(X_test_pca_cnn)
y_pred_labels_pca = y_pred_probs_pca.argmax(axis=1)
y_test_labels_pca = y_test_pca_cnn.argmax(axis=1)

acc_cnn_pca = accuracy_score(y_test_labels_pca, y_pred_labels_pca)
prec_cnn_pca = precision_score(y_test_labels_pca, y_pred_labels_pca,
                                average="macro", zero_division=0)
rec_cnn_pca = recall_score(y_test_labels_pca, y_pred_labels_pca,
                            average="macro", zero_division=0)
f1_cnn_pca = f1_score(y_test_labels_pca, y_pred_labels_pca,
                       average="macro", zero_division=0)

print("\n==== CNN After PCA ===")
print("Accuracy:", acc_cnn_pca)
print("Precision:", prec_cnn_pca)
print("Recall:", rec_cnn_pca)
print("F1:", f1_cnn_pca)
print("\nConfusion Matrix:\n",
      confusion_matrix(y_test_labels_pca, y_pred_labels_pca))

```

## Overall Effect of PCA

Comparing the results before and after PCA, both models lose performance: KNN drops from roughly 0.75–0.86 accuracy down to about 0.57, and the CNN falls from around 0.86 accuracy to roughly 0.43. This makes sense in our setting:

- PCA keeps directions of maximum variance, which do not always align with the directions that best separate Offensive and Defensive roles.
- With only six engineered features, the original space is already low-dimensional and informative, so compressing everything into two principal components mostly discards useful information.

Overall, PCA does not help classification accuracy for this specific dataset, but it confirms that our hand-crafted efficiency features already capture most of the structure needed for distinguishing offensive and defensive archetypes.

## References

1. EE016 Course Jupyter Notebooks:

- *Correlation.ipynb*
- *Visualizing\_EE16.ipynb*
- *Data Loading and Preprocessing Notebook*

2. EE016 Project Instructions:

- Dataset Further Analysis and Visualization Phase
- Model Training and Evaluation Guidelines

3. Seaborn and Matplotlib Documentation:

- <https://seaborn.pydata.org>
- <https://matplotlib.org>

4. Pandas Documentation:

- <https://pandas.pydata.org>