Ann

September 22, 2024

0.0.1 Problem Statement

With so many players and so much data, it can be difficult to form a cohesive team.

Loom Links:

- Teddy Coon: Watch Video
- Owen Kroeger: Watch Video

0.0.2 Algorithm

1. **Import Libraries**
   - Import necessary libraries for data handling, model training, and visualization.
2. **Load Player Performance Data**
   - Load data from a CSV file containing NBA player performance metrics.
3. **Filter Data**
   - Filter players based on the desired seasons (2015-2019).
   - Randomly select a subset of 100 players from the filtered data.
4. **Calculate Role Scores**
   - Define a function to calculate various performance metrics (e.g., scorer, playmaker, rebounder, defender, utility).
   - Apply this function to the filtered DataFrame.
5. **Prepare Data for Model Training**
   - Standardize the data to balance feature scales.
   - Convert the data to PyTorch's preferred format for training.
6. **Define Neural Network Model**
   - Create a class for a role-specific Multi-Layer Perceptron (MLP) model.
7. **Train the Model for Each Player Type**
   - Split the data into training and testing sets.
   - Instantiate the neural network model.
   - Define the loss function and optimizer.
   - Train the model for each role (scorer, playmaker, rebounder, defender, utility).
8. **Make Predictions and Evaluate**
   - Make predictions on the entire dataset.
   - Store the results in the DataFrame.
9. **Select the Optimal Team**
   - Run through loops to evaluate players and select the optimal team.
   - Ensure that each role has a unique player assigned.

10. **Visualize Results**
    - Print the optimal team.
    - Display heatmaps showing player evaluations for each position.

[25]:
```
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

All imports are shown above

[26]:
```
data_path = 'all_season.csv'
players_df = pd.read_csv(data_path)
```

Loading the dataset

[15]:
```
selected_years = ['2015-16', '2016-17', '2017-18', '2018-19', '2019-20']
pool_df = players_df[players_df['season'].isin(selected_years)]
```

Filtering the dataframe: only includes players who played between 2015-2016 and 2019-2020

[27]:
```
pool_df = pool_df.sample(n=100, random_state=42)
```

This piece of code randomly selects 100 players from the filtered dataset.

[28]:
```
def calculate_role_scores(df):
    df['scorer_score'] = df['pts']
    df['playmaker_score'] = df['ast']
    df['rebounder_score'] = df['reb']
    df['defender_score'] = df['net_rating']
    df['utility_score'] = (df['pts'] + df['reb'] + df['ast']) / 3  # Average␣
 ↪instead of sum
    df['impact_score'] = df['usg_pct'] * df['net_rating']
    df['size_factor'] = df['player_height'] * df['player_weight']
    return df
```

0.0.3 Role Score Calculation Function

This function takes in information on the data frame and creates new columns that reflect specific aspects of the basketball player's performance. For example:

- **Points (pts)**: Used to calculate the scoring ability.
- **Assists**: Contribute to the playmaking score.

Additionally, we created more advanced statistics like the **Utility Score**, which is a combination of the key performance metrics: scoring, rebounding, and assists.s

[29]:
```
scored_df = calculate_role_scores(pool_df)
```

This function is straightforward just preparing the data using the function that we created above

[30]:
```
features = scored_df[['scorer_score', 'playmaker_score', 'rebounder_score',
                      'defender_score', 'utility_score', 'impact_score',␣
 ↪'size_factor']].values
scaler = StandardScaler()
features_scaled = scaler.fit_transform(features)
X_tensor = torch.tensor(features_scaled, dtype=torch.float32)
class RoleSpecificMLP(nn.Module):
    def __init__(self, input_size, hidden_sizes):
        super(RoleSpecificMLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(input_size, hidden_sizes[0]),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(hidden_sizes[0], hidden_sizes[1]),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(hidden_sizes[1], 1)  # Single output for ranking
        )
    def forward(self, x):
        return self.layers(x)
input_size = X_tensor.shape[1]
hidden_sizes = [128, 64]
```

0.0.4 Data Preparation for Neural Network

In this step, we continue preparing the data for the neural network:

- **Features**: A dataset containing the raw data for the 100 players selected within the specified timeframe.
- **Scaler**: An object created to standardize the features by scaling the data to have a mean of 0 and a standard deviation of 1.
- **PyTorch Tensor**: We use `torch.tensor` to convert the scaled data into a format suitable for the PyTorch libraries, preparing it for model training.

[20]:
```
def train_role_model(X_tensor, target_score, role_name):
    y_tensor = torch.tensor(target_score, dtype=torch.float32).view(-1, 1)
    X_train, X_test, y_train, y_test = train_test_split(X_tensor, y_tensor,␣
 ↪test_size=0.2, random_state=42)
    model = RoleSpecificMLP(input_size, hidden_sizes)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    epochs = 100
    for epoch in range(epochs):
```

```
        model.train()
        outputs = model(X_train)
        loss = criterion(outputs, y_train)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (epoch + 1) % 10 == 0:
            print(f'{role_name} - Epoch [{epoch + 1}/{epochs}], Loss: {loss.
↪item():.4f}')

    model.eval()
    with torch.no_grad():
        predictions = model(X_tensor).numpy().flatten()
        scored_df[f'{role_name}_predicted_score'] = predictions

    grid_scores = np.array(predictions).reshape(10, 10)
    return grid_scores, scored_df[[f'{role_name}_predicted_score',␣
↪'player_name', 'team_abbreviation', 'scorer_score',
                                   'playmaker_score', 'rebounder_score',␣
↪'defender_score', 'utility_score']]
```

0.0.5 Role-Specific Model Training Function

This function begins by converting the target scores into a PyTorch tensor and reshaping it for compatibility with the model. The steps include:

- **Data Splitting**: The function splits the input feature tensor into training and testing sets, using 80% of the data for training and 20% for evaluation.
- **Model Instantiation**: An instance of the `RoleSpecificMLP` model is created.
- **Loss Function and Optimizer**: The Mean Squared Error (MSE) loss function is defined alongside the Adam (Adaptive Movement Estimation) optimizer, which efficiently updates model parameters.
- **Training Process**:
  - The model is trained over 100 epochs, where it learns to minimize the loss between predicted and actual scores through forward and backward passes.
  - The function logs the loss every ten epochs to monitor the model's performance.
- **Model Evaluation and Prediction**:
  - After training, the model evaluates the entire feature dataset, generating predictions without gradient tracking.
  - These predictions are stored in the original DataFrame alongside relevant player information.
- **Grid Visualization**:

– The predictions are reshaped into a 10x10 grid format, making it easier to visualize and analyze player performance in various roles.

The function returns both the grid scores and the updated DataFrame, providing valuable insights into player evaluations for different team roles.

[21]:
```
scorer_grid, scorer_candidates = train_role_model(X_tensor,␣
 ↪scored_df['scorer_score'].values, 'Scorer')
playmaker_grid, playmaker_candidates = train_role_model(X_tensor,␣
 ↪scored_df['playmaker_score'].values, 'Playmaker')
rebounder_grid, rebounder_candidates = train_role_model(X_tensor,␣
 ↪scored_df['rebounder_score'].values, 'Rebounder')
defender_grid, defender_candidates = train_role_model(X_tensor,␣
 ↪scored_df['defender_score'].values, 'Defender')
utility_grid, utility_candidates = train_role_model(X_tensor,␣
 ↪scored_df['utility_score'].values, 'Utility')
```

```
Scorer - Epoch [10/100], Loss: 69.2450
Scorer - Epoch [20/100], Loss: 60.8631
Scorer - Epoch [30/100], Loss: 48.8092
Scorer - Epoch [40/100], Loss: 37.5109
Scorer - Epoch [50/100], Loss: 21.9623
Scorer - Epoch [60/100], Loss: 11.6625
Scorer - Epoch [70/100], Loss: 8.8604
Scorer - Epoch [80/100], Loss: 7.9728
Scorer - Epoch [90/100], Loss: 5.6992
Scorer - Epoch [100/100], Loss: 4.8779
Playmaker - Epoch [10/100], Loss: 4.2449
Playmaker - Epoch [20/100], Loss: 2.4884
Playmaker - Epoch [30/100], Loss: 1.4023
Playmaker - Epoch [40/100], Loss: 0.8348
Playmaker - Epoch [50/100], Loss: 0.4783
Playmaker - Epoch [60/100], Loss: 0.4905
Playmaker - Epoch [70/100], Loss: 0.5204
Playmaker - Epoch [80/100], Loss: 0.3776
Playmaker - Epoch [90/100], Loss: 0.3297
Playmaker - Epoch [100/100], Loss: 0.3164
Rebounder - Epoch [10/100], Loss: 10.2974
Rebounder - Epoch [20/100], Loss: 6.8086
Rebounder - Epoch [30/100], Loss: 3.8228
Rebounder - Epoch [40/100], Loss: 2.5727
Rebounder - Epoch [50/100], Loss: 2.2688
Rebounder - Epoch [60/100], Loss: 2.6731
Rebounder - Epoch [70/100], Loss: 1.3648
Rebounder - Epoch [80/100], Loss: 1.0631
Rebounder - Epoch [90/100], Loss: 0.9590
Rebounder - Epoch [100/100], Loss: 0.7201
Defender - Epoch [10/100], Loss: 136.0559
```

```
Defender - Epoch [20/100], Loss: 126.0754
Defender - Epoch [30/100], Loss: 110.5732
Defender - Epoch [40/100], Loss: 84.1980
Defender - Epoch [50/100], Loss: 59.1214
Defender - Epoch [60/100], Loss: 39.7896
Defender - Epoch [70/100], Loss: 13.2981
Defender - Epoch [80/100], Loss: 8.5108
Defender - Epoch [90/100], Loss: 12.8226
Defender - Epoch [100/100], Loss: 14.0132
Utility - Epoch [10/100], Loss: 19.5950
Utility - Epoch [20/100], Loss: 14.4654
Utility - Epoch [30/100], Loss: 9.6951
Utility - Epoch [40/100], Loss: 4.9855
Utility - Epoch [50/100], Loss: 3.6340
Utility - Epoch [60/100], Loss: 2.4590
Utility - Epoch [70/100], Loss: 2.2629
Utility - Epoch [80/100], Loss: 2.2283
Utility - Epoch [90/100], Loss: 1.7817
Utility - Epoch [100/100], Loss: 1.2712
```

Here we call the model to train for all 5 of the roles on our team

[22]:
```
all_candidates = pd.concat([scorer_candidates, playmaker_candidates,␣
 ↪rebounder_candidates, defender_candidates, utility_candidates])
assigned_players = set()
optimal_team = []
role_order = [
    ('Scorer', scorer_candidates),
    ('Playmaker', playmaker_candidates),
    ('Rebounder', rebounder_candidates),
    ('Defender', defender_candidates),
    ('Utility', utility_candidates)
]
selected_positions = {}
for role_name, candidates in role_order:
    candidates = candidates.sort_values(by=f'{role_name}_predicted_score',␣
 ↪ascending=False)
    for _, player in candidates.iterrows():
        if player['player_name'] not in assigned_players:
            player['predicted_role'] = role_name
            optimal_team.append(player)
            assigned_players.add(player['player_name'])
            grid_index = candidates.index.get_loc(player.name)
            selected_positions[role_name] = (grid_index // 10, grid_index % 10,␣
 ↪player['player_name'])
            break
```

0.0.6 Optimal Team Selection Code Explanation

This code snippet systematically selects players for specific basketball roles to form an optimal team based on predicted scores. The process is as follows:

- **Role Definition and Candidate Sorting**:
  - An ordered list of roles and their corresponding candidate DataFrames is defined.
  - For each role, candidates are sorted by their predicted scores in descending order, ensuring the highest-rated players are prioritized.
- **Player Assignment Loop**:
  - The inner loop iterates through the sorted candidates, checking if a player has already been assigned a role.
  - If a player is available, they are assigned the current role, added to the `optimal_team`, and marked as assigned to prevent duplicates.
  - The loop breaks after selecting one player for each role, ensuring a single assignment per position.
- **Final Optimal Team Composition**:
  - This process continues for all defined roles, resulting in an optimal team composed of the best players for each specific role.
  - The approach effectively leverages the predicted performance metrics, ensuring the team is composed of players who are best suited for each position.

[23]:
```
optimal_team_df = pd.DataFrame(optimal_team)
print("\nOptimal 5-Man Team Based on Separate Role-Specific Models (with Unique␣
 ↪Selections):")
print(optimal_team_df[['player_name', 'team_abbreviation', 'scorer_score',␣
 ↪'playmaker_score',
                       'rebounder_score', 'defender_score', 'utility_score',␣
 ↪'predicted_role']])
```

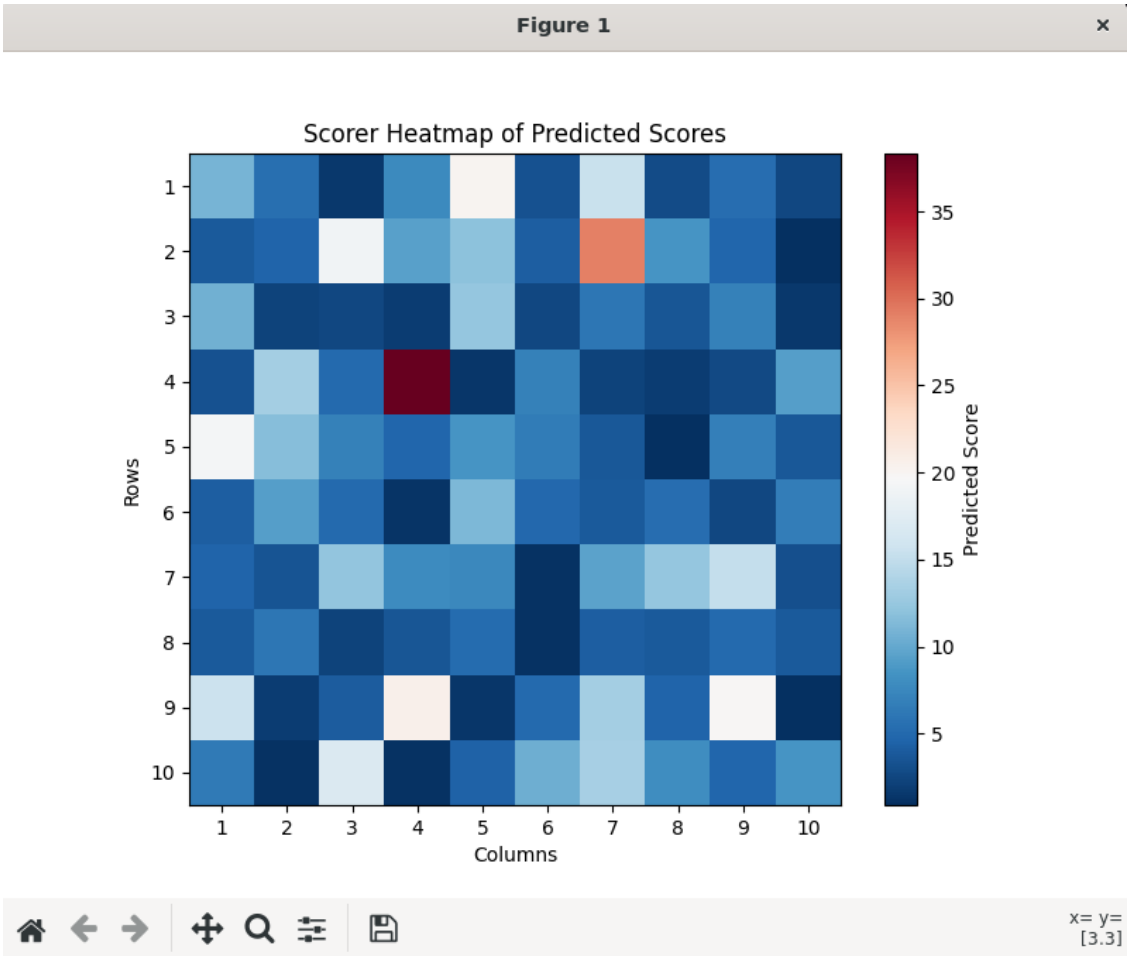
```
Optimal 5-Man Team Based on Separate Role-Specific Models (with Unique
Selections):
                player_name team_abbreviation  scorer_score  playmaker_score  \
10674  Giannis Antetokounmpo               MIL          29.5              5.6
10177           Jeff Teague               MIN          12.1              8.2
9358        DeMarcus Cousins               NOP          27.0              4.6
10703         Henry Ellenson               BKN           0.4              0.2
9955             Ben Simmons               PHI          15.8              8.2

       rebounder_score  defender_score  utility_score predicted_role
10674             13.6            15.4      16.233333         Scorer
10177              2.5            -0.1       7.600000      Playmaker
9358              11.0            -2.2      14.200000      Rebounder
10703              1.2            27.3       0.600000       Defender
9955               8.1             6.6      10.700000        Utility
```

0.0.7 Optimal Team DataFrame Creation and Display

This code section creates a DataFrame, `optimal_team_df`, from the list of selected players for the optimal basketball team.

- **Team Composition**:
  - The code prints a message indicating the display of the optimal five-man team.
  - It then outputs specific columns, including player names, team abbreviations, various role-related scores, and the assigned predicted roles.
- **Purpose**:
  - This structured presentation helps visualize the team's composition.
  - It highlights each player's performance metrics in their respective roles, providing clear insights into why each player was selected.



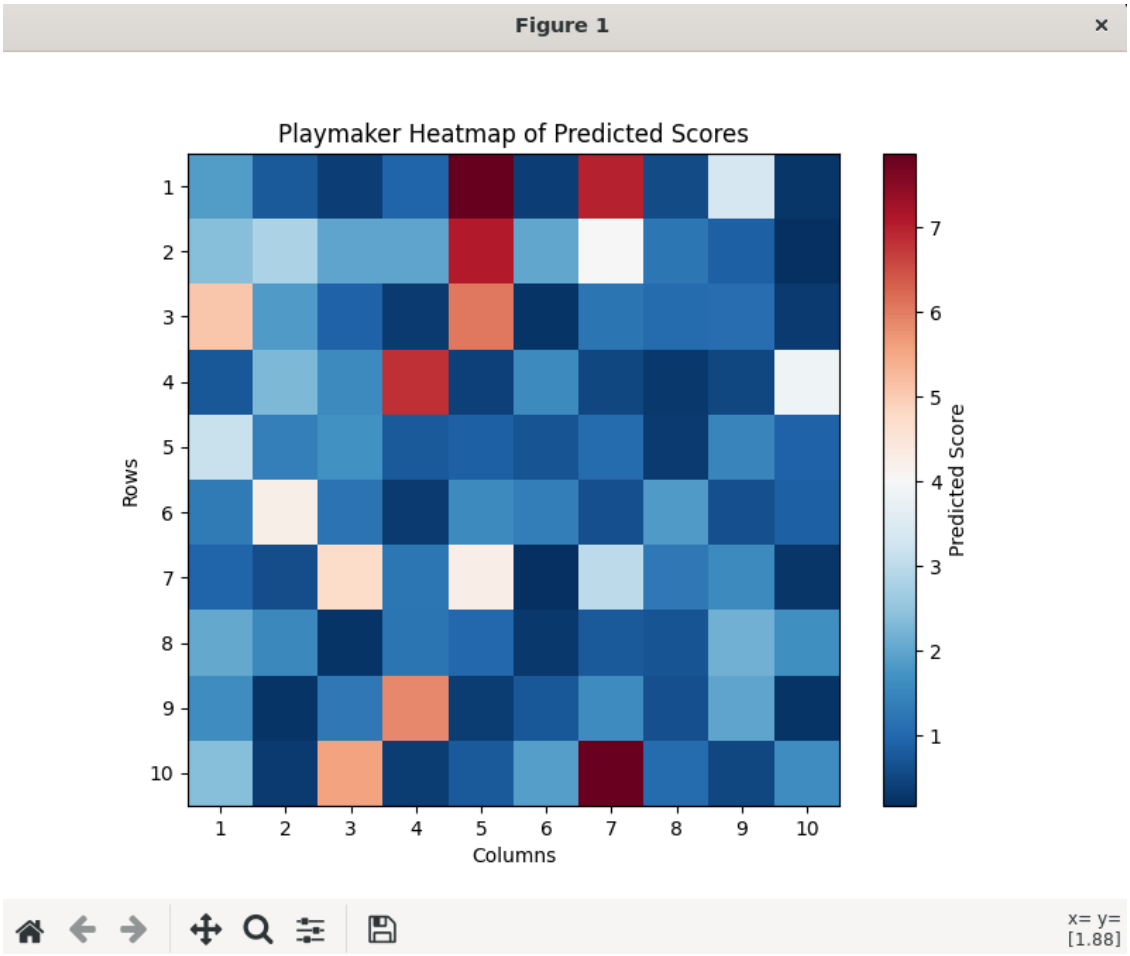


Figure 1
Playmaker Heatmap of Predicted Scores
Rows
Columns
Predicted Score
1
2
3
4
5
6
7
8
9
10
x= y=
[1.88]

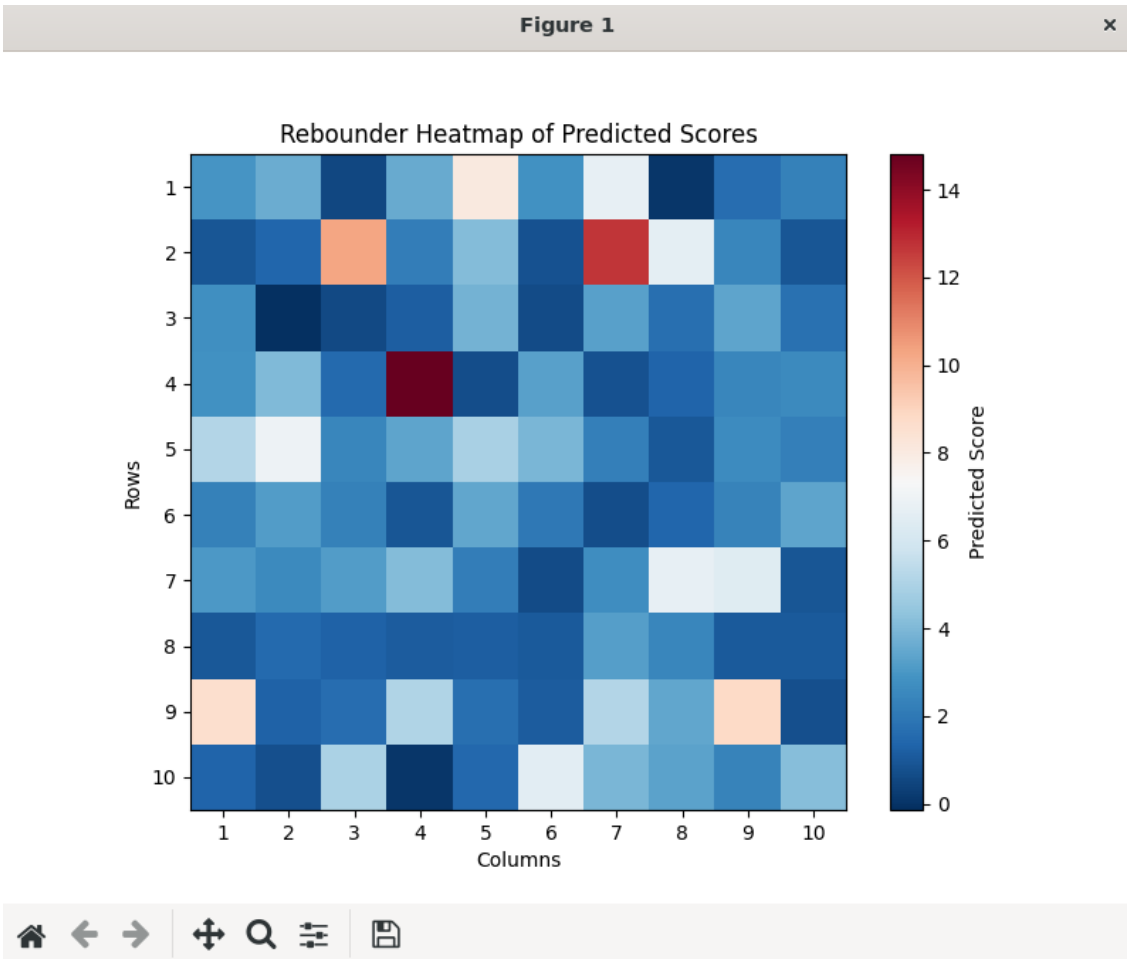


Figure 1
Rebounder Heatmap of Predicted Scores
Rows
Columns
Predicted Score
1
2
3
4
5
6
7
8
9
10
0
2
4
6
8
10
12
14

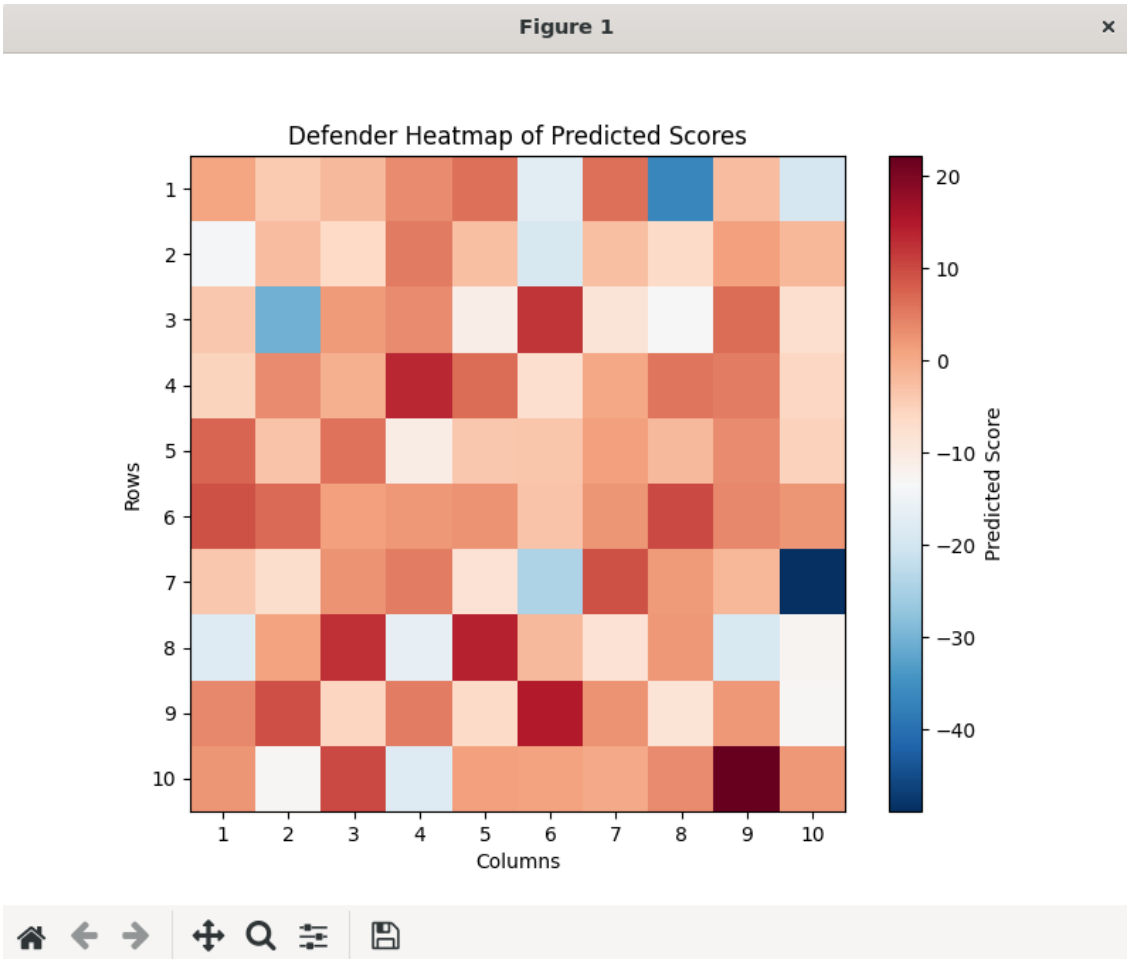
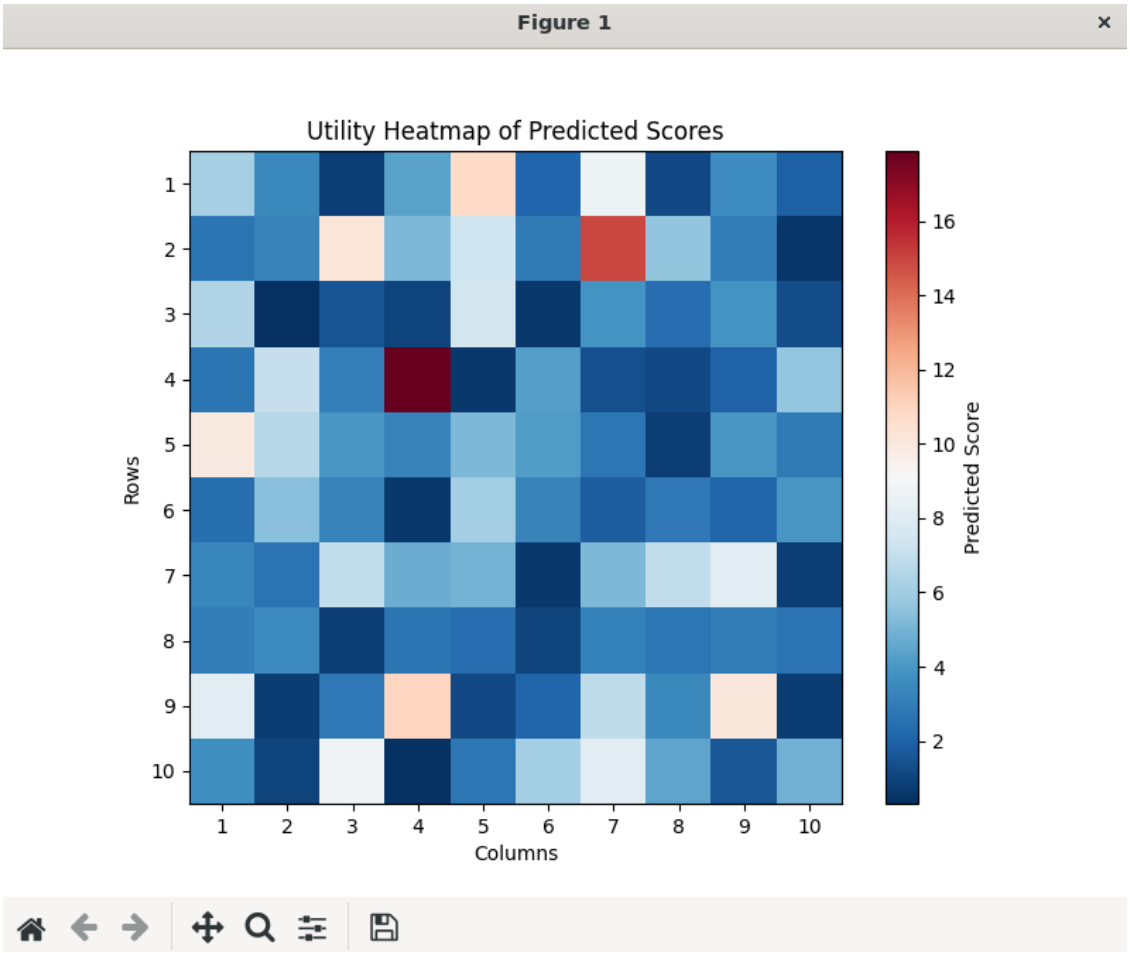


Figure 1
Defender Heatmap of Predicted Scores
Rows
Columns
Predicted Score
1
2
3
4
5
6
7
8
9
10
20
10
0
−10
−20
−30
−40



[ ]:
```
# Function to annotate and display heatmaps with the selected players
def plot_heatmap_with_selection(grid, role_name, candidates_df):
    plt.figure(figsize=(8, 6))
    im = plt.imshow(grid, cmap='RdBu_r', interpolation='nearest')
    plt.colorbar(label='Predicted Score')
    plt.title(f'{role_name} Heatmap of Predicted Scores')
    plt.xlabel('Columns')
    plt.ylabel('Rows')
    plt.xticks(ticks=np.arange(10), labels=np.arange(1, 11))
    plt.yticks(ticks=np.arange(10), labels=np.arange(1, 11))

    # Annotate the heatmap with the selected player's name
    selected_player = optimal_team_df[optimal_team_df['predicted_role'] ==␣
 ↪role_name]
    for _, player in selected_player.iterrows():
        grid_index = candidates_df[candidates_df['player_name'] ==␣
 ↪player['player_name']].index[0]
        row, col = divmod(grid_index, 10)
        plt.text(col, row, player['player_name'], ha='center', va='center',
```

```
                color='black', fontweight='bold', bbox=dict(facecolor='white',␣
↪alpha=0.7, edgecolor='none'))

    plt.show()

# Plot each heatmap with the correct final selected player names
plot_heatmap_with_selection(scorer_grid, 'Scorer', scorer_candidates)
plot_heatmap_with_selection(playmaker_grid, 'Playmaker', playmaker_candidates)
plot_heatmap_with_selection(rebounder_grid, 'Rebounder', rebounder_candidates)
plot_heatmap_with_selection(defender_grid, 'Defender', defender_candidates)
plot_heatmap_with_selection(utility_grid, 'Utility', utility_candidates)
```

The code provided above creates heat maps for each of the 5 positions displaying of the players that we tested. With darker reds showing a higher score and blues being lower scores.