NBA Prediction Model

Akhil Gorla, Aditya Iyer

Collecting/Processing Data

Webscraping

```python
import requests
from bs4 import BeautifulSoup
import pandas as pd
import time
import os

BASE_URL = 'https://www.basketball-reference.com'
team_urls = ['/teams/ATL/', '/teams/BOS/', '/teams/CHA/', '/teams/CHI/', '/teams/CLE/', '/tea

google_drive_dir = "/content/drive/MyDrive/2023 summer project"

def save_table_as_csv(table, team_dir):
    df = pd.read_html(str(table))[0]

    caption_tag = table.find('caption')
    if caption_tag:
        table_name = caption_tag.get_text().strip().replace(' ', '_').replace('.', '')
    else:
        table_name = "table"

    filename = table_name + '.csv'

    save_dir = os.path.join(google_drive_dir, team_dir)
    os.makedirs(save_dir, exist_ok=True)
    file_handle = os.path.join(save_dir, filename)

    df.to_csv(file_handle, index=False)
    print(f"Saved {file_handle}...")

def get_gamelog_for_team(team_url, year):
    print(f"Attempting to extract data for {team_url} for the {year}-{year+1} season")
    url = BASE_URL + team_url + str(year) + "/gamelog/"

    try:
        response = requests.get(url)
        response.raise_for_status()

        soup = BeautifulSoup(response.text, 'html.parser')
        table = soup.find('table', {'id': 'tgl_basic'})
        if not table:
            print(f"No table found for URL: {url}")
            return None
```

- Automates collection of basketball games logs, stores in CSV files
- Modified and normalized to feed to neural network
- Contains every game from 1980 - 2023

Script Constants and Imports

```
import requests
from bs4 import BeautifulSoup
import pandas as pd
import time
import os

BASE_URL = 'https://www.basketball-reference.com'
team_urls = ['/teams/ATL/', '/teams/BOS/', '/teams/CHA/', '/teams/CHI/'

google_drive_dir = "/content/drive/MyDrive/2023 summer project"
```

- **Requests:** Fetch HTML content of web pages.
- **BeautifulSoup:** Parse HTML content and extract relevant data.
- **Pandas:** Manipulate and save data in CSV format.
- **OS:** Manage file and directory operations.
- **Time:** Pause execution to avoid server overload.

- **BASE_URL:** The base URL for Basketball Reference.
- **team_urls:** List of team-specific URLs.
- **google_drive_dir:** Directory for saving the CSV files.

Save Tables as CSV

```python
def save_table_as_csv(table, team_dir):
    df = pd.read_html(str(table))[0]

    caption_tag = table.find('caption')
    if caption_tag:
        table_name = caption_tag.get_text().strip().replace(' ', '_').replace('.', '')
    else:
        table_name = "table"

    filename = table_name + '.csv'

    save_dir = os.path.join(google_drive_dir, team_dir)
    os.makedirs(save_dir, exist_ok=True)
    file_handle = os.path.join(save_dir, filename)
```

Purpose: Save DataFrame as CSV in the specified directory.

Parameters: DataFrame, team directory, and table name.

Operations: Create directory if not exists, save DataFrame as CSV.

Get Game Logs for a Team and Season

```
def get_gamelog_for_team(team_url, year):
    print(f"Attempting to extract data for {team_url} for the {year}-{year+1} season")
    url = BASE_URL + team_url + str(year) + "/gamelog/"

    try:
        response = requests.get(url)
        response.raise_for_status()

        soup = BeautifulSoup(response.text, 'html.parser')
        table = soup.find('table', {'id': 'tgl_basic'})
        if not table:
            print(f"No table found for URL: {url}")
            return None

        df = pd.read_html(str(table))[0]
        df['Season'] = f"{year}-{year+1}"

        # Save the dataframe to CSV
        team_name = team_url.split("/")[2]
        save_dir = f"{team_name}_{year}-{year+1}"
        save_table_as_csv(table, save_dir)

        return df

    except requests.RequestException as e:
        print(f"Error fetching {url}. Error: {e}")
        return None
```

Purpose: Fetch and process game log data for a specific team and season.

Operations:

- Construct URL.
- Send request and parse HTML.
- Extract data into DataFrame.
- Add Season and team_name columns.
- Save DataFrame as CSV.

Main Loop to Iterate Over Teams and Years

```
all_dataframes = []

for year in range(1960, 2003):
    for team_url in team_urls:
        df = get_gamelog_for_team(team_url, year)
        if df is not None:
            all_dataframes.append(df)
        time.sleep(2)

combined_df = pd.concat(all_dataframes, ignore_index=True)
print(combined_df)
```

- **Purpose:** Loop through each team and season to collect game logs.
- **Operations:**
  - Call get_gamelog_for_team function.
  - Append DataFrame to list if not None.
  - Pause for 2 seconds between requests.
  - Combine all Data Frames into a single DataFrame.
  - Save combined DataFrame as a CSV.

Results and Output

Content:

- **Combined DataFrame:** A single Data Frame containing game logs for all teams and seasons.
- **Output CSV:** combined_gamelogs.csv saved in the specified directory.
- **Sample Output:**

	Rk	G	Date	home_or_away	Opp	W/L	Tm	Opp_2	FG	FGA	...	FT%_2	ORB_2	TRB	AST_2	STL_2	BLK_2	TOV_2	PF_2	Season	team_name
0	1	1	1959-10-17	NaN	CIN	W	129	125	50	114	...	.846	NaN	64	21	NaN	NaN	NaN	33	1959-1960	BOS
1	4	4	1959-11-01	@	CIN	W	124	109	45	NaN	...	.853	NaN	59	NaN	NaN	NaN	NaN	37	1959-1960	BOS
2	5	5	1959-11-03	@	STL	W	103	98	38	109	...	.500	NaN	80	NaN	NaN	NaN	NaN	29	1959-1960	BOS
3	6	6	1959-11-07	NaN	PHW	W	115	106	43	107	...	.585	NaN	73	13	NaN	NaN	NaN	25	1959-1960	BOS
4	8	8	1959-11-10	N	DET	W	128	109	51	117	...	.700	NaN	60	NaN	NaN	NaN	NaN	32	1959-1960	BOS

Data Cleaning and Manipulation

Issues:

1. Each game was recorded twice.
2. Thousands of null values due to several stats not being recorded until 1979.
3. Multiple data-types in each column

Due to these issues and more, We decided to reformat the data completely to make it easier to create features and feed them into the Neural network.

New Sample Data:

	Game Date	Home Team	Away Team	Home Team Score	Away Team Score	Home_W/L	Away_W/L	Home_Team_FG	Home_Team_FGA	Away_Team_FT%	Away_Team_ORB	Away_Team_TRB	Away_Team_AST	Away_Team_STL	Away_Team_BLK
0	2002-10-31	ATL	UTA	105	98	W	L	39	76	.750	11	36.0	19	6	3
1	2002-11-02	ATL	CHI	98	92	W	L	34	80	.735	10	36.0	15	8	1
2	2002-11-18	ATL	TOR	117	92	W	L	49	86	.813	8	42.0	30	6	3
3	2002-11-20	ATL	MIN	93	103	L	W	34	85	.692	16	44.0	18	8	3
4	2002-11-23	ATL	BOS	99	109	L	W	27	69	.704	10	35.0	22	6	3

Feature Creation

```python
def update_elo(winner_elo, loser_elo, K=20):
    expected_winner = 1 / (1 + 10 ** ((loser_elo - winner_elo) / 400))
    expected_loser = 1 - expected_winner
    new_winner_elo = winner_elo + K * (1 - expected_winner)
    new_loser_elo = loser_elo + K * (0 - expected_loser)
    return new_winner_elo, new_loser_elo
```

```python
import pandas as pd


data['Game Date'] = pd.to_datetime(data['Game Date'])
data.sort_values(['Home Team', 'Game Date'], inplace=True)
columns_to_average = [
    'Home Team Score', 'Away Team Score', 'Home_Team_FG', 'Home_Team_FGA', 'Home_Team_FG%',
    'Home_Team_3P', 'Home_Team_3PA', 'Home_Team_3P%', 'Home_Team_FT', 'Home_Team_FTA', 'Home_Team_FT%',
    'Home_Team_ORB', 'Home_Team_TRB', 'Home_Team_AST', 'Home_Team_STL', 'Home_Team_BLK', 'Home_Team_TOV',
    'Home_Team_PF', 'Away_Team_FG', 'Away_Team_FGA', 'Away_Team_FG%', 'Away_Team_3P', 'Away_Team_3PA',
    'Away_Team_3P%', 'Away_Team_FT', 'Away_Team_FTA', 'Away_Team_FT%', 'Away_Team_ORB', 'Away_Team_TRB',
    'Away_Team_AST', 'Away_Team_STL', 'Away_Team_BLK', 'Away_Team_TOV', 'Away_Team_PF'
]


for col in columns_to_average:
    data[f'RA_{col}'] = data.groupby('Home Team')[col].transform(lambda x: x.shift(1).rolling(window=5).mean())
```

There are two types of features that we created

1. Elo Rating - A numerical rating that denotes the strength of the team based on the strengths of the teams they won or lost against.

2. The second type of feature is the counting stats for each team over the past 5 games averaged in order to avoid data leakage while still giving the model a good sense of how that team is producing in that statistic

Neural Network Architecture

```python
# Assuming the data and feature_columns are defined
target_column = 'Home_W/L_W'  # This column is 1 for win, 0 otherwise

X = data[feature_columns]
y = data[target_column].values

# Standardizing features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Convert to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)

# Create TensorDatasets and DataLoaders
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

Standardize and Split Data

```python
class NBAPredictionModel(nn.Module):
    def __init__(self):
        super(NBAPredictionModel, self).__init__()
        self.fc1 = nn.Linear(len(feature_columns), 256)
        self.bn1 = nn.BatchNorm1d(256)
        self.dropout = nn.Dropout(0.7)
        self.fc2 = nn.Linear(256, 128)
        self.bn2 = nn.BatchNorm1d(128)
        self.fc3 = nn.Linear(128, 64)
        self.bn3 = nn.BatchNorm1d(64)
        self.fc4 = nn.Linear(64, 32)
        self.bn4 = nn.BatchNorm1d(32)
        self.fc5 = nn.Linear(32, 1)

    def forward(self, x):
        x = F.relu(self.bn1(self.fc1(x)))
        x = self.dropout(x)
        x = F.relu(self.bn2(self.fc2(x)))
        x = F.relu(self.bn3(self.fc3(x)))
        x = self.dropout(x)
        x = F.relu(self.bn4(self.fc4(x)))
        x = torch.sigmoid(self.fc5(x))
        return x
```

```python
class NBAPredictionModel(nn.Module):
    def __init__(self):
        super(NBAPredictionModel, self).__init__()
        self.fc1 = nn.Linear(len(feature_columns), 128)
        self.bn1 = nn.BatchNorm1d(128)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(128, 64)
        self.bn2 = nn.BatchNorm1d(64)
        self.fc3 = nn.Linear(64, 32)
        self.bn3 = nn.BatchNorm1d(32)
        self.fc4 = nn.Linear(32, 16)
        self.bn4 = nn.BatchNorm1d(16)
        self.fc5 = nn.Linear(16, 1)

    def forward(self, x):
        x = F.relu(self.bn1(self.fc1(x)))
        x = self.dropout(x)
        x = F.relu(self.bn2(self.fc2(x)))
        x = F.relu(self.bn3(self.fc3(x)))
        x = self.dropout(x)
        x = F.relu(self.bn4(self.fc4(x)))
        x = torch.sigmoid(self.fc5(x))
        return x
```

Creating the Neural Network

Choosing the right Optimizer, Learning Rate, Decay

```
# Initialize model, loss function, and optimizer
model = NBAPredictionModel()
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0005, weight_decay=1e-5)  # L2 regularization
```

Considerations:

- Learning rate vs the number of epochs
- Weight decay to prevent overfitting for generalization

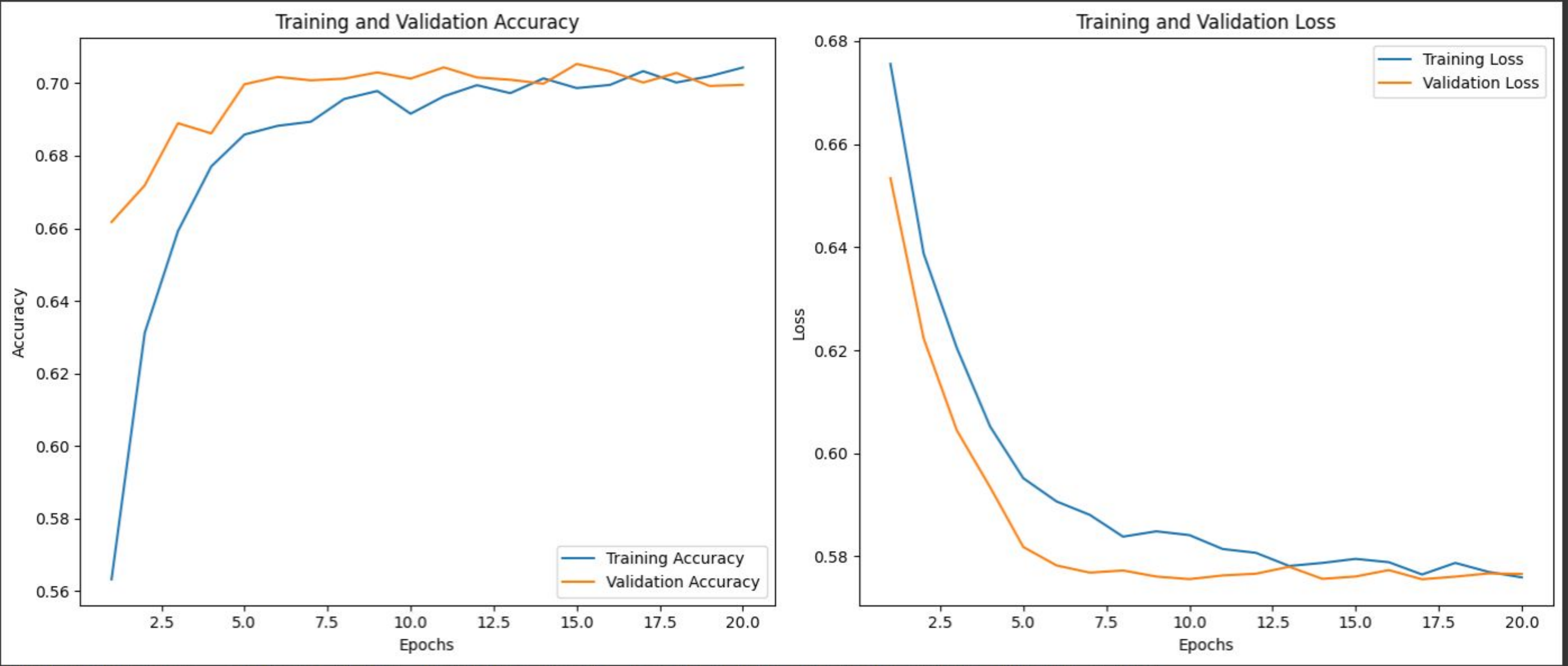
- Track running stats for all metrics to evaluate model performance
- Change epochs of training in relation to learning rate to prevent overfitting

```
for epoch in range(num_epochs):
    model.train()
    train_true_labels = []
    train_predictions = []
    running_train_loss = 0.0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs.squeeze(), labels)
        loss.backward()
        optimizer.step()

        # Store predictions and true labels for training accu
        predicted = (outputs > 0.5).float()
        train_true_labels.extend(labels.numpy())
        train_predictions.extend(predicted.detach().numpy())

        running_train_loss += loss.item() * inputs.size(0)
```

```
# Evaluate the model on the validation set
model.eval()
val_true_labels = []
val_predictions = []
running_val_loss = 0.0
with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs).squeeze()
        loss = criterion(outputs, labels)
        predicted = (outputs > 0.5).float()
        val_true_labels.extend(labels.numpy())
        val_predictions.extend(predicted.numpy())
        running_val_loss += loss.item() * inputs.size(0)

val_accuracy = accuracy_score(val_true_labels, val_predictions)
val_accuracies.append(val_accuracy)
val_loss = running_val_loss / len(test_loader.dataset)
val_losses.append(val_loss)
```

Running and Evaluating Model

Evaluating Model



Final Test Metrics - Accuracy: 0.6995, Precision: 0.7506, Recall: 0.7402, F1 Score: 0.7454, AUC-ROC: 0.6901

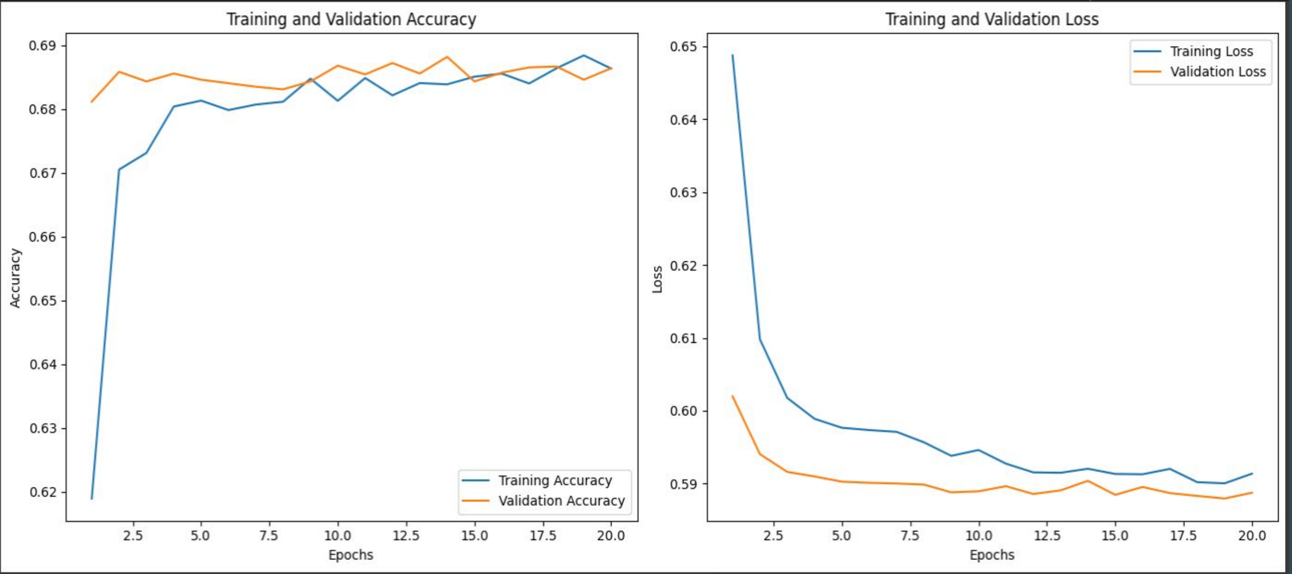
Evaluating Model: Cross-Validation

```python
# K-Fold Cross-Validation
k_folds = 5
kfold = KFold(n_splits=k_folds, shuffle=True, random_state=42)
fold_results = []
```

```
Fold 5/5
Epoch 1/20, Loss: 0.6578852534294128, Train Accuracy: 0.5670, Val Accuracy: 0.6779, Train Loss: 0.6760, Val Loss: 0.6377
Epoch 2/20, Loss: 0.619010329246521, Train Accuracy: 0.6418, Val Accuracy: 0.6949, Train Loss: 0.6348, Val Loss: 0.6019
Epoch 3/20, Loss: 0.6030265092849731, Train Accuracy: 0.6651, Val Accuracy: 0.6977, Train Loss: 0.6106, Val Loss: 0.5867
Epoch 4/20, Loss: 0.5944220423698425, Train Accuracy: 0.6790, Val Accuracy: 0.7017, Train Loss: 0.6013, Val Loss: 0.5795
Epoch 5/20, Loss: 0.5861063599586487, Train Accuracy: 0.6884, Val Accuracy: 0.7040, Train Loss: 0.5930, Val Loss: 0.5770
Epoch 6/20, Loss: 0.5881703495979309, Train Accuracy: 0.6902, Val Accuracy: 0.7031, Train Loss: 0.5886, Val Loss: 0.5776
Epoch 7/20, Loss: 0.5792232751846313, Train Accuracy: 0.6943, Val Accuracy: 0.7012, Train Loss: 0.5847, Val Loss: 0.5747
Epoch 8/20, Loss: 0.5759660005569458, Train Accuracy: 0.6958, Val Accuracy: 0.7033, Train Loss: 0.5841, Val Loss: 0.5734
Epoch 9/20, Loss: 0.5834667086601257, Train Accuracy: 0.6983, Val Accuracy: 0.6991, Train Loss: 0.5795, Val Loss: 0.5759
Epoch 10/20, Loss: 0.5747405886650085, Train Accuracy: 0.6968, Val Accuracy: 0.7026, Train Loss: 0.5818, Val Loss: 0.5744
Epoch 11/20, Loss: 0.5713751316070557, Train Accuracy: 0.7002, Val Accuracy: 0.7010, Train Loss: 0.5789, Val Loss: 0.5731
Epoch 12/20, Loss: 0.5711924433708191, Train Accuracy: 0.7013, Val Accuracy: 0.6996, Train Loss: 0.5774, Val Loss: 0.5736
Epoch 13/20, Loss: 0.577813982963562, Train Accuracy: 0.7010, Val Accuracy: 0.7019, Train Loss: 0.5755, Val Loss: 0.5728
Epoch 14/20, Loss: 0.5774003267288208, Train Accuracy: 0.7057, Val Accuracy: 0.6979, Train Loss: 0.5743, Val Loss: 0.5736
Epoch 15/20, Loss: 0.574280858039856, Train Accuracy: 0.7007, Val Accuracy: 0.6965, Train Loss: 0.5771, Val Loss: 0.5740
Epoch 16/20, Loss: 0.5754866003990173, Train Accuracy: 0.7029, Val Accuracy: 0.6963, Train Loss: 0.5758, Val Loss: 0.5754
Epoch 17/20, Loss: 0.5702217817306519, Train Accuracy: 0.7024, Val Accuracy: 0.6998, Train Loss: 0.5758, Val Loss: 0.5734
Epoch 18/20, Loss: 0.5719321966171265, Train Accuracy: 0.7034, Val Accuracy: 0.7005, Train Loss: 0.5738, Val Loss: 0.5743
Epoch 19/20, Loss: 0.5745109915733337, Train Accuracy: 0.7016, Val Accuracy: 0.6991, Train Loss: 0.5747, Val Loss: 0.5730
Epoch 20/20, Loss: 0.5730219483375549, Train Accuracy: 0.7056, Val Accuracy: 0.6989, Train Loss: 0.5728, Val Loss: 0.5734
Fold 5 Results - Accuracy: 0.6989, Precision: 0.7201, Recall: 0.8005, F1 Score: 0.7582, AUC-ROC: 0.6767, Loss: 0.5734
Average Results across 5 folds - Accuracy: 0.7056, Precision: 0.7324, Recall: 0.7926, F1 Score: 0.7612, AUC-ROC: 0.6859, Loss: 0.5708
```

Adding Data

```
[ ] final_data = pd.read_csv(os.path.join(str(google_drive_dir), 'Final_data.csv'))
    final_data.rename(columns={' Home_W/L ': 'Home_W/L', ' Away_W/L ': 'Away_W/L'}, inplace=True)
    final_data = pd.get_dummies(final_data, columns=['Home_W/L', 'Away_W/L'])
```




```
Final Test Metrics - Accuracy: 0.6864, Precision: 0.7041, Recall: 0.8289, F1 Score: 0.7614, AUC-ROC: 0.6492
```

Issues:

- Need to adjust Neural Network/Weights for the new data
- Old data has inconsistencies due to the game changing over time

Key Takeaways

Ways we can Improve the model:

- Minimize loss
- Fix possible overfitting
- Add more scaled data
- Utilize other metrics to improve model

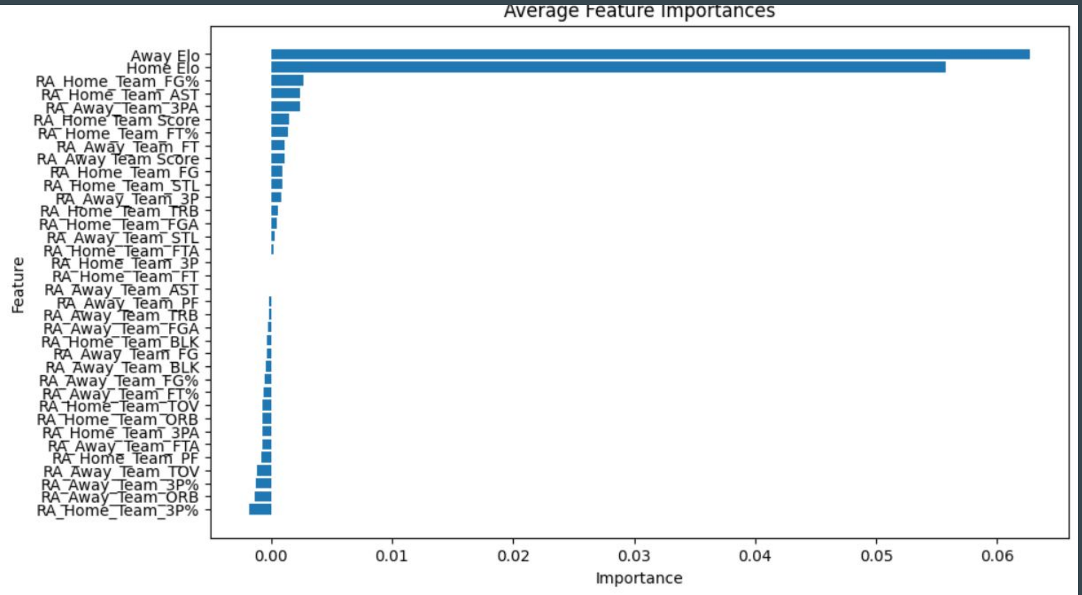
Best Metrics Achieved (Independent):

- Accuracy: .7283
- Precision: .7545
- Recall: 0.8594
- F1 Score: 0.7760
- AUC-ROC Score: 0.7079
- Loss: 0.4748*, 0.5301**

Things to learn more about:

- The real statistics behind each of the various functions
- More regularization techniques
- Identifying and fixing anomalies
- Assumptions required to study data

Exploratory analysis




```
Feature: Home Elo, Importance: 0.0998
Feature: Away Elo, Importance: 0.0717
Feature: RA_Home Team Score, Importance: -0.0001
Feature: RA_Away Team Score, Importance: -0.0038
Feature: RA_Home_Team_FG, Importance: -0.0015
Feature: RA_Home_Team_FGA, Importance: -0.0020
Feature: RA_Home_Team_FG%, Importance: 0.0001
Feature: RA_Home_Team_3P, Importance: 0.0001
Feature: RA_Home_Team_3PA, Importance: -0.0009
Feature: RA_Home_Team_3P%, Importance: -0.0022
Feature: RA_Home_Team_FT, Importance: 0.0000
Feature: RA_Home_Team_FTA, Importance: -0.0033
Feature: RA_Home_Team_FT%, Importance: -0.0017
Feature: RA_Home_Team_ORB, Importance: -0.0001
Feature: RA_Home_Team_TRB, Importance: -0.0022
Feature: RA_Home_Team_AST, Importance: -0.0011
Feature: RA_Home_Team_STL, Importance: -0.0010
Feature: RA_Home_Team_BLK, Importance: 0.0006
Feature: RA_Home_Team_TOV, Importance: -0.0013
Feature: RA_Home_Team_PF, Importance: -0.0009
Feature: RA_Away_Team_FG, Importance: -0.0015
Feature: RA_Away_Team_FGA, Importance: -0.0006
Feature: RA_Away_Team_FG%, Importance: -0.0008
Feature: RA_Away_Team_3P, Importance: -0.0030
```

Possible Effects to Explore:

- Feature Importance
- Metric Analysis in terms of the data

Links

Neural Network/Testing

Web Scraping/Processing/Initial & Old Models