Predicting NCAA March Madness Matchup Outcomes

Comp490

Final Project

Joseph Guidi & Billy Winters

Problem introduction:

For years fans, developers, and AI have been trying to predict the perfect March Madness NCAA bracket. Some have come close but nobody has ever been perfect. Every year 64 teams compete in a single elimination bracket to be crowned March Madness Champion. They are ranked from 1 seed to 16 seed in four different divisions. The goal of this model is to predict the winner of each match up throughout the tournament.

Proposed Solution:

1. The first solution that we decided to look at is a single layer neural network using TensorFlow.
2. The second solution that we decided to look at is a multi-layer neural network using TensorFlow
3. The third solution that we decided to look at is using XGboost.

Import Libraries and CSV files

In [1]:
```
#import pandas, numpy, tensorflow, and sklearn
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split

#for XGboost
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score

#"C:\Users\josep\Downloads\Rand_outcomes_2023.csv"

#Load the data
outcomes = pd.read_csv("C:/Users/josep/Downloads/Rand_outcomes_2023.csv")
perfect = pd.read_csv("C:/Users/josep/Downloads/Perfect2023.csv")
```

Experimental Setup:

The data

Outline

The data used was obtained from a kaggle dataset called "March Machine Learning Mania 2023" presented by Kaggle(https://www.kaggle.com/competitions/march-machine-learning-mania-2023/overview). There were 31 csv files which we pulled a total of 13 features out of. There was over 20 years of data in these csv files however for the limited time we had for this project we only learned about the previous season, 2023.

Team Features

Column Name	Description
TeamID	Unique Numeric Value
TotalWins	Total wins for a team in 2023 season
TotalLoses	Total losses for a team in 2023 season
TotalPF	Total points for a team in 2023 season
TotalPA	Total points against a team in 2023 season
TotalFGA	Total field goals attempted as team in 2023 season
TotalFGM	Total field goals made as team in 2023 season
TotalFGA3	Total field goals attempted, 3's, as team in 2023 season
TotalFGM3	Total field goals Made, 3's, as team in 2023 season
TotalFTA	Total free throws attempted as team in 2023 season
TotalFTM	Total free throws made as team in 2023 season
TotalStl	Total steals as team in 2023 season
TotalBlk	Total blocks as team in 2023 season

Each team had these 12 features which made up the inputs of the neural network.

Training Matchup Outcomes

TeamA	TeamB	Outcome
1101	1238	1
1319	1101	0
1477	1101	0
1101	1167	1
1101	1470	1
1101	1426	1
1469	1101	0

Each matchup from the 2023 regular season was listed on a csv file and the outcome was also provided, 1 if TeamA wins, 0 if TeamA loses. This made up the output of the neural network, 1/0.

1. In order to set up a single layer neural network we had to massage the massive amounts of data that we were given. After a few hours of working I was able to pull out the correct 12 features needed to train the neural network. Then input the data into a signle layer consisting of 24 nodes to an output layer consisting of 1 node resulting in a 1/0.

2. In order to set up a multi layer neural network we had to massage the massive amounts of data that we were given. After a few hours of working I was able to pull out the correct 12 features needed to train the neural network. Then input the data into a 24 node large layer through a 24 node hidden layer down to a 1 node output layer printing a 1/0.

3. In order to set up a solution using XGboost we had to massage the massive amounts of data that we were given. After a few hours of working I was able to pull out the correct 12 features needed to train the neural network.

In [2]:
```
perfect.head(5)
```

Out[2]:

	TeamID	TeamName	Confrence	TotalWins	TotalLosses	TotalPF	TotalPA	TotalFGA	TotalFGM
0	1101	Abilene Chr	Southland Conference	8	17	1829	1952	1431	634
1	1102	Air Force	Western Athletic Conference	14	18	2199	2239	1722	780
2	1103	Akron	Ohio Valley Conference	20	11	2317	2114	1793	810
3	1104	Alabama	Southeastern Conference	29	5	2699	2345	2096	941
4	1105	Alabama A&M	Southwest Athletic Conference	12	18	1987	2064	1713	704

In [3]:
```
outcomes.head(5)
```

Out[3]:

	TeamA	TeamB	Outcome
0	1101	1238	1
1	1319	1101	0
2	1477	1101	0
3	1101	1167	1
4	1101	1470	1

Merge the matchups

The inputs from each team need to to be merged together so therefore they can be listed into the neural network as head to head matchups.

In [4]:
```
# Merge the two datasets
outcomes_perfect = outcomes.merge(perfect, left_on='TeamA', right_on='TeamID').merge(p

# Create input and output arrays
X = outcomes_perfect[['TotalWins_A', 'TotalLosses_A', 'TotalPF_A', 'TotalPA_A', 'Total
y = outcomes_perfect['Outcome'].to_numpy()
outcomes_perfect.head(5)
```

Out[4]:

	TeamA	TeamB	Outcome	TeamID_A	TeamName_A	Confrence_A	TotalWins_A	TotalLosses_A	Tota
0	1101	1238	1	1101	Abilene Chr	Southland Conference	8	17	
1	1290	1238	0	1290	MS Valley St	Southwest Athletic Conference	5	21	
2	1290	1238	1	1290	MS Valley St	Southwest Athletic Conference	5	21	
3	1103	1238	1	1103	Akron	Ohio Valley Conference	20	11	
4	1104	1238	1	1104	Alabama	Southeastern Conference	29	5	

5 rows × 33 columns

The Neural Network (Run Only NN Solution #1 or #2 or XGBoost Solution #3)

Tensorflow was the driving force behind the neural network. There was no need to split the data into training and test data because we are testing the neural network on the tournament matchups so therefore test data is already classified.

In [5]:
```
X_train, X_cv, y_train, y_cv = train_test_split(X, y, test_size=0.2, random_state=42)
```

Solution #1 Single Layer

In [6]:
```
# Define the model
model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(1, input_shape=(24,), activation='sigmoid')
])

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Solution #2 Multi-Layer

In [6]:
```
# Define the model
model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(24, input_shape=(24,), activation='relu'),
  tf.keras.layers.Dense(1, activation='sigmoid')
])
```

```
# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

In [4]:
```
#performed poorly around 50% accuracy
'''
# Define the model
model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(24, input_shape=(24,), activation='relu'),
  tf.keras.layers.Dense(8, activation='relu'),
  tf.keras.layers.Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
'''

#This is a crazy unrealistic model thought that I'd try it
# helped to realize an error in data I believe
'''
model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(64, input_shape=(24,), activation='relu'),
  tf.keras.layers.Dense(32, activation='relu'),
  tf.keras.layers.Dense(16, activation='relu'),
  tf.keras.layers.Dense(32, activation='relu'),
  tf.keras.layers.Dense(64, activation='relu'),
  tf.keras.layers.Dense(1, activation='sigmoid')
])
'''
```

XGBoost

An XGBoost model is trained using a specific dataset and a set of hyperparameters that define its behavior. The model consists of an ensemble of decision trees, specifically gradient boosted trees. During the training process, XGBoost sequentially builds these trees, with each subsequent tree attempting to correct the errors made by the previous trees.

Solution #3 XGBoost (Skip "Training Time")

In [6]:
```
#Define and fit the model

model = XGBClassifier()
history = model.fit(X_train, y_train)

y_pred = model.predict(X)
predictions = [round(value) for value in y_pred]

accuracy = accuracy_score(y, predictions)
print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

```
Accuracy: 89.22%
```

Training Time

In [7]:
```
# Train the model
history = model.fit(X_train, y_train, epochs=150, batch_size=32, validation_data=(X_cv
```

```
Epoch 1/150
141/141 [==============================] - 1s 4ms/step - loss: 114.9898 - accuracy:
0.5497 - val_loss: 10.2012 - val_accuracy: 0.5727
Epoch 2/150
141/141 [==============================] - 0s 2ms/step - loss: 9.8662 - accuracy: 0.5
521 - val_loss: 8.3168 - val_accuracy: 0.5888
Epoch 3/150
141/141 [==============================] - 0s 2ms/step - loss: 7.9801 - accuracy: 0.5
637 - val_loss: 6.3480 - val_accuracy: 0.5995
Epoch 4/150
141/141 [==============================] - 0s 2ms/step - loss: 6.1313 - accuracy: 0.5
876 - val_loss: 8.1543 - val_accuracy: 0.5486
Epoch 5/150
141/141 [==============================] - 0s 2ms/step - loss: 5.0613 - accuracy: 0.6
121 - val_loss: 4.6898 - val_accuracy: 0.6111
Epoch 6/150
141/141 [==============================] - 0s 2ms/step - loss: 4.4535 - accuracy: 0.6
213 - val_loss: 5.2786 - val_accuracy: 0.5879
Epoch 7/150
141/141 [==============================] - 0s 2ms/step - loss: 4.3859 - accuracy: 0.6
231 - val_loss: 10.7773 - val_accuracy: 0.5103
Epoch 8/150
141/141 [==============================] - 0s 2ms/step - loss: 4.6772 - accuracy: 0.6
150 - val_loss: 3.2553 - val_accuracy: 0.6351
Epoch 9/150
141/141 [==============================] - 0s 2ms/step - loss: 3.6955 - accuracy: 0.6
237 - val_loss: 6.8458 - val_accuracy: 0.5522
Epoch 10/150
141/141 [==============================] - 0s 2ms/step - loss: 3.8610 - accuracy: 0.6
213 - val_loss: 2.6750 - val_accuracy: 0.6441
Epoch 11/150
141/141 [==============================] - 0s 2ms/step - loss: 3.2357 - accuracy: 0.6
322 - val_loss: 4.0682 - val_accuracy: 0.5932
Epoch 12/150
141/141 [==============================] - 0s 2ms/step - loss: 3.4598 - accuracy: 0.6
309 - val_loss: 2.8025 - val_accuracy: 0.6334
Epoch 13/150
141/141 [==============================] - 0s 2ms/step - loss: 2.9812 - accuracy: 0.6
360 - val_loss: 6.0970 - val_accuracy: 0.5424
Epoch 14/150
141/141 [==============================] - 0s 2ms/step - loss: 3.1282 - accuracy: 0.6
249 - val_loss: 2.8046 - val_accuracy: 0.6351
Epoch 15/150
141/141 [==============================] - 0s 2ms/step - loss: 2.6755 - accuracy: 0.6
360 - val_loss: 9.3012 - val_accuracy: 0.5085
Epoch 16/150
141/141 [==============================] - 0s 2ms/step - loss: 4.0006 - accuracy: 0.6
048 - val_loss: 3.6141 - val_accuracy: 0.5888
Epoch 17/150
141/141 [==============================] - 0s 2ms/step - loss: 2.5649 - accuracy: 0.6
470 - val_loss: 2.2947 - val_accuracy: 0.6289
Epoch 18/150
141/141 [==============================] - 0s 2ms/step - loss: 2.4673 - accuracy: 0.6
273 - val_loss: 1.9439 - val_accuracy: 0.6601
Epoch 19/150
141/141 [==============================] - 0s 2ms/step - loss: 2.7913 - accuracy: 0.6
202 - val_loss: 2.2954 - val_accuracy: 0.6450
Epoch 20/150
141/141 [==============================] - 0s 2ms/step - loss: 2.4346 - accuracy: 0.6
340 - val_loss: 9.3313 - val_accuracy: 0.5049
```

```
Epoch 21/150
141/141 [==============================] - 0s 2ms/step - loss: 3.1707 - accuracy: 0.6
217 - val_loss: 2.3971 - val_accuracy: 0.6227
Epoch 22/150
141/141 [==============================] - 0s 2ms/step - loss: 2.4182 - accuracy: 0.6
282 - val_loss: 2.9456 - val_accuracy: 0.5852
Epoch 23/150
141/141 [==============================] - 0s 2ms/step - loss: 2.2909 - accuracy: 0.6
447 - val_loss: 2.6830 - val_accuracy: 0.5986
Epoch 24/150
141/141 [==============================] - 0s 2ms/step - loss: 2.1558 - accuracy: 0.6
387 - val_loss: 9.9862 - val_accuracy: 0.4924
Epoch 25/150
141/141 [==============================] - 0s 2ms/step - loss: 3.0864 - accuracy: 0.6
106 - val_loss: 8.1486 - val_accuracy: 0.4942
Epoch 26/150
141/141 [==============================] - 0s 2ms/step - loss: 3.3497 - accuracy: 0.6
112 - val_loss: 1.9402 - val_accuracy: 0.6512
Epoch 27/150
141/141 [==============================] - 0s 2ms/step - loss: 1.8410 - accuracy: 0.6
389 - val_loss: 1.9648 - val_accuracy: 0.6441
Epoch 28/150
141/141 [==============================] - 0s 3ms/step - loss: 1.9297 - accuracy: 0.6
367 - val_loss: 2.7586 - val_accuracy: 0.5736
Epoch 29/150
141/141 [==============================] - 0s 3ms/step - loss: 2.1106 - accuracy: 0.6
304 - val_loss: 1.6850 - val_accuracy: 0.6450
Epoch 30/150
141/141 [==============================] - 0s 2ms/step - loss: 1.6888 - accuracy: 0.6
494 - val_loss: 8.3745 - val_accuracy: 0.5076
Epoch 31/150
141/141 [==============================] - 0s 2ms/step - loss: 2.7418 - accuracy: 0.6
282 - val_loss: 2.3357 - val_accuracy: 0.5870
Epoch 32/150
141/141 [==============================] - 0s 2ms/step - loss: 1.8942 - accuracy: 0.6
302 - val_loss: 4.3693 - val_accuracy: 0.5040
Epoch 33/150
141/141 [==============================] - 0s 2ms/step - loss: 2.4085 - accuracy: 0.6
130 - val_loss: 2.4164 - val_accuracy: 0.6012
Epoch 34/150
141/141 [==============================] - 0s 2ms/step - loss: 1.9940 - accuracy: 0.6
383 - val_loss: 3.9893 - val_accuracy: 0.5192
Epoch 35/150
141/141 [==============================] - 0s 2ms/step - loss: 1.7778 - accuracy: 0.6
383 - val_loss: 1.2329 - val_accuracy: 0.6690
Epoch 36/150
141/141 [==============================] - 0s 2ms/step - loss: 1.7720 - accuracy: 0.6
349 - val_loss: 1.3689 - val_accuracy: 0.6673
Epoch 37/150
141/141 [==============================] - 0s 2ms/step - loss: 1.5466 - accuracy: 0.6
465 - val_loss: 6.5798 - val_accuracy: 0.4933
Epoch 38/150
141/141 [==============================] - 0s 2ms/step - loss: 1.8544 - accuracy: 0.6
309 - val_loss: 1.0984 - val_accuracy: 0.6798
Epoch 39/150
141/141 [==============================] - 0s 2ms/step - loss: 1.7139 - accuracy: 0.6
284 - val_loss: 1.4386 - val_accuracy: 0.6343
Epoch 40/150
141/141 [==============================] - 0s 2ms/step - loss: 1.5462 - accuracy: 0.6
345 - val_loss: 4.8750 - val_accuracy: 0.4996
```

```
Epoch 41/150
141/141 [==============================] - 0s 2ms/step - loss: 1.9316 - accuracy: 0.6
316 - val_loss: 1.2090 - val_accuracy: 0.6708
Epoch 42/150
141/141 [==============================] - 0s 2ms/step - loss: 1.3520 - accuracy: 0.6
548 - val_loss: 1.0674 - val_accuracy: 0.6628
Epoch 43/150
141/141 [==============================] - 0s 2ms/step - loss: 1.2837 - accuracy: 0.6
492 - val_loss: 1.4622 - val_accuracy: 0.6405
Epoch 44/150
141/141 [==============================] - 0s 2ms/step - loss: 1.5323 - accuracy: 0.6
358 - val_loss: 1.3370 - val_accuracy: 0.6271
Epoch 45/150
141/141 [==============================] - 0s 2ms/step - loss: 1.3403 - accuracy: 0.6
432 - val_loss: 11.5591 - val_accuracy: 0.4933
Epoch 46/150
141/141 [==============================] - 0s 2ms/step - loss: 2.5566 - accuracy: 0.6
284 - val_loss: 3.3067 - val_accuracy: 0.5343
Epoch 47/150
141/141 [==============================] - 0s 2ms/step - loss: 1.7896 - accuracy: 0.6
309 - val_loss: 2.1462 - val_accuracy: 0.5789
Epoch 48/150
141/141 [==============================] - 0s 2ms/step - loss: 1.7009 - accuracy: 0.6
213 - val_loss: 7.5659 - val_accuracy: 0.4933
Epoch 49/150
141/141 [==============================] - 0s 2ms/step - loss: 1.9086 - accuracy: 0.6
298 - val_loss: 2.7046 - val_accuracy: 0.5442
Epoch 50/150
141/141 [==============================] - 0s 3ms/step - loss: 1.4264 - accuracy: 0.6
550 - val_loss: 0.9756 - val_accuracy: 0.6762
Epoch 51/150
141/141 [==============================] - 0s 2ms/step - loss: 1.4927 - accuracy: 0.6
438 - val_loss: 4.2484 - val_accuracy: 0.5156
Epoch 52/150
141/141 [==============================] - 0s 2ms/step - loss: 1.5022 - accuracy: 0.6
505 - val_loss: 2.3010 - val_accuracy: 0.5415
Epoch 53/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1396 - accuracy: 0.6
583 - val_loss: 5.1948 - val_accuracy: 0.5094
Epoch 54/150
141/141 [==============================] - 0s 2ms/step - loss: 1.7392 - accuracy: 0.6
139 - val_loss: 1.6034 - val_accuracy: 0.5950
Epoch 55/150
141/141 [==============================] - 0s 3ms/step - loss: 1.4679 - accuracy: 0.6
472 - val_loss: 1.8283 - val_accuracy: 0.5888
Epoch 56/150
141/141 [==============================] - 0s 3ms/step - loss: 1.2536 - accuracy: 0.6
628 - val_loss: 3.5166 - val_accuracy: 0.5174
Epoch 57/150
141/141 [==============================] - 0s 2ms/step - loss: 1.3382 - accuracy: 0.6
396 - val_loss: 2.5350 - val_accuracy: 0.5450
Epoch 58/150
141/141 [==============================] - 0s 2ms/step - loss: 1.3142 - accuracy: 0.6
371 - val_loss: 4.1089 - val_accuracy: 0.5138
Epoch 59/150
141/141 [==============================] - 0s 2ms/step - loss: 1.6670 - accuracy: 0.6
213 - val_loss: 1.9292 - val_accuracy: 0.5798
Epoch 60/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1166 - accuracy: 0.6
724 - val_loss: 2.6237 - val_accuracy: 0.5183
```

```
Epoch 61/150
141/141 [==============================] - 0s 2ms/step - loss: 1.2519 - accuracy: 0.6
443 - val_loss: 1.1464 - val_accuracy: 0.6539
Epoch 62/150
141/141 [==============================] - 0s 2ms/step - loss: 1.3137 - accuracy: 0.6
423 - val_loss: 2.7121 - val_accuracy: 0.5183
Epoch 63/150
141/141 [==============================] - 0s 2ms/step - loss: 1.5381 - accuracy: 0.6
311 - val_loss: 1.6685 - val_accuracy: 0.5941
Epoch 64/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1726 - accuracy: 0.6
581 - val_loss: 1.0104 - val_accuracy: 0.6521
Epoch 65/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0415 - accuracy: 0.6
568 - val_loss: 7.2889 - val_accuracy: 0.4933
Epoch 66/150
141/141 [==============================] - 0s 2ms/step - loss: 1.4866 - accuracy: 0.6
472 - val_loss: 0.7991 - val_accuracy: 0.7074
Epoch 67/150
141/141 [==============================] - 0s 3ms/step - loss: 1.3280 - accuracy: 0.6
340 - val_loss: 1.9907 - val_accuracy: 0.5718
Epoch 68/150
141/141 [==============================] - 0s 3ms/step - loss: 1.3002 - accuracy: 0.6
456 - val_loss: 2.8223 - val_accuracy: 0.5335
Epoch 69/150
141/141 [==============================] - 0s 2ms/step - loss: 1.2501 - accuracy: 0.6
396 - val_loss: 2.8743 - val_accuracy: 0.5272
Epoch 70/150
141/141 [==============================] - 0s 2ms/step - loss: 1.4725 - accuracy: 0.6
338 - val_loss: 0.7837 - val_accuracy: 0.7092
Epoch 71/150
141/141 [==============================] - 0s 2ms/step - loss: 1.2181 - accuracy: 0.6
523 - val_loss: 5.3225 - val_accuracy: 0.5067
Epoch 72/150
141/141 [==============================] - 0s 2ms/step - loss: 1.3271 - accuracy: 0.6
570 - val_loss: 1.1192 - val_accuracy: 0.6441
Epoch 73/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1374 - accuracy: 0.6
557 - val_loss: 4.9429 - val_accuracy: 0.4960
Epoch 74/150
141/141 [==============================] - 0s 2ms/step - loss: 1.4714 - accuracy: 0.6
412 - val_loss: 0.7609 - val_accuracy: 0.7226
Epoch 75/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1889 - accuracy: 0.6
476 - val_loss: 4.6272 - val_accuracy: 0.5103
Epoch 76/150
141/141 [==============================] - 0s 2ms/step - loss: 1.3543 - accuracy: 0.6
456 - val_loss: 7.0777 - val_accuracy: 0.5076
Epoch 77/150
141/141 [==============================] - 0s 2ms/step - loss: 2.3893 - accuracy: 0.6
170 - val_loss: 1.6725 - val_accuracy: 0.5834
Epoch 78/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1925 - accuracy: 0.6
572 - val_loss: 4.4842 - val_accuracy: 0.5103
Epoch 79/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1689 - accuracy: 0.6
561 - val_loss: 4.0220 - val_accuracy: 0.5112
Epoch 80/150
141/141 [==============================] - 0s 2ms/step - loss: 1.5118 - accuracy: 0.6
255 - val_loss: 1.9193 - val_accuracy: 0.5638
```

```
Epoch 81/150
141/141 [==============================] - 0s 2ms/step - loss: 1.2970 - accuracy: 0.6
396 - val_loss: 0.7974 - val_accuracy: 0.7154
Epoch 82/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0032 - accuracy: 0.6
748 - val_loss: 3.8038 - val_accuracy: 0.5112
Epoch 83/150
141/141 [==============================] - 0s 2ms/step - loss: 1.4275 - accuracy: 0.6
403 - val_loss: 4.1955 - val_accuracy: 0.4960
Epoch 84/150
141/141 [==============================] - 0s 2ms/step - loss: 1.3425 - accuracy: 0.6
423 - val_loss: 0.8020 - val_accuracy: 0.7172
Epoch 85/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1785 - accuracy: 0.6
458 - val_loss: 6.8802 - val_accuracy: 0.4942
Epoch 86/150
141/141 [==============================] - 0s 2ms/step - loss: 1.6178 - accuracy: 0.6
536 - val_loss: 3.7001 - val_accuracy: 0.4969
Epoch 87/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1886 - accuracy: 0.6
565 - val_loss: 7.2264 - val_accuracy: 0.5067
Epoch 88/150
141/141 [==============================] - 0s 2ms/step - loss: 1.7678 - accuracy: 0.6
298 - val_loss: 3.7738 - val_accuracy: 0.5147
Epoch 89/150
141/141 [==============================] - 0s 2ms/step - loss: 1.3474 - accuracy: 0.6
670 - val_loss: 3.7719 - val_accuracy: 0.5120
Epoch 90/150
141/141 [==============================] - 0s 2ms/step - loss: 1.2121 - accuracy: 0.6
574 - val_loss: 2.2086 - val_accuracy: 0.5245
Epoch 91/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1761 - accuracy: 0.6
458 - val_loss: 0.7789 - val_accuracy: 0.6905
Epoch 92/150
141/141 [==============================] - 0s 2ms/step - loss: 1.2056 - accuracy: 0.6
534 - val_loss: 1.5326 - val_accuracy: 0.5852
Epoch 93/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1894 - accuracy: 0.6
615 - val_loss: 1.5088 - val_accuracy: 0.5754
Epoch 94/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0443 - accuracy: 0.6
590 - val_loss: 7.8771 - val_accuracy: 0.4933
Epoch 95/150
141/141 [==============================] - 0s 2ms/step - loss: 1.9197 - accuracy: 0.6
400 - val_loss: 2.9046 - val_accuracy: 0.5165
Epoch 96/150
141/141 [==============================] - 0s 2ms/step - loss: 1.2254 - accuracy: 0.6
487 - val_loss: 1.9104 - val_accuracy: 0.5477
Epoch 97/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0942 - accuracy: 0.6
646 - val_loss: 1.0760 - val_accuracy: 0.6325
Epoch 98/150
141/141 [==============================] - 0s 2ms/step - loss: 0.9363 - accuracy: 0.6
737 - val_loss: 3.7534 - val_accuracy: 0.4942
Epoch 99/150
141/141 [==============================] - 0s 2ms/step - loss: 1.4284 - accuracy: 0.6
485 - val_loss: 3.4014 - val_accuracy: 0.5022
Epoch 100/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1305 - accuracy: 0.6
536 - val_loss: 0.7325 - val_accuracy: 0.7226
```

```
Epoch 101/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0730 - accuracy: 0.6
519 - val_loss: 1.7784 - val_accuracy: 0.5638
Epoch 102/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1034 - accuracy: 0.6
565 - val_loss: 0.7150 - val_accuracy: 0.7190
Epoch 103/150
141/141 [==============================] - 0s 2ms/step - loss: 0.9310 - accuracy: 0.6
603 - val_loss: 0.9090 - val_accuracy: 0.6369
Epoch 104/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1951 - accuracy: 0.6
441 - val_loss: 2.1310 - val_accuracy: 0.5290
Epoch 105/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0538 - accuracy: 0.6
570 - val_loss: 4.5796 - val_accuracy: 0.5094
Epoch 106/150
141/141 [==============================] - 0s 2ms/step - loss: 1.5805 - accuracy: 0.6
211 - val_loss: 0.7825 - val_accuracy: 0.7056
Epoch 107/150
141/141 [==============================] - 0s 2ms/step - loss: 0.9988 - accuracy: 0.6
684 - val_loss: 1.0418 - val_accuracy: 0.6432
Epoch 108/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0364 - accuracy: 0.6
570 - val_loss: 5.0498 - val_accuracy: 0.4933
Epoch 109/150
141/141 [==============================] - 0s 2ms/step - loss: 1.4991 - accuracy: 0.6
525 - val_loss: 1.4166 - val_accuracy: 0.6084
Epoch 110/150
141/141 [==============================] - 0s 2ms/step - loss: 1.4495 - accuracy: 0.6
485 - val_loss: 0.8010 - val_accuracy: 0.7003
Epoch 111/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1475 - accuracy: 0.6
467 - val_loss: 3.6240 - val_accuracy: 0.4969
Epoch 112/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0007 - accuracy: 0.6
603 - val_loss: 3.3796 - val_accuracy: 0.5138
Epoch 113/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0291 - accuracy: 0.6
635 - val_loss: 1.4623 - val_accuracy: 0.5709
Epoch 114/150
141/141 [==============================] - 0s 2ms/step - loss: 0.8774 - accuracy: 0.6
724 - val_loss: 0.6579 - val_accuracy: 0.7288
Epoch 115/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1287 - accuracy: 0.6
483 - val_loss: 1.4206 - val_accuracy: 0.5781
Epoch 116/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0702 - accuracy: 0.6
541 - val_loss: 1.2476 - val_accuracy: 0.6048
Epoch 117/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0778 - accuracy: 0.6
557 - val_loss: 5.2992 - val_accuracy: 0.4942
Epoch 118/150
141/141 [==============================] - 0s 2ms/step - loss: 1.4255 - accuracy: 0.6
521 - val_loss: 0.9278 - val_accuracy: 0.6467
Epoch 119/150
141/141 [==============================] - 0s 2ms/step - loss: 0.9520 - accuracy: 0.6
706 - val_loss: 1.6488 - val_accuracy: 0.5638
Epoch 120/150
141/141 [==============================] - 0s 2ms/step - loss: 0.9677 - accuracy: 0.6
635 - val_loss: 2.0669 - val_accuracy: 0.5236
```

```
Epoch 121/150
141/141 [==============================] - 0s 2ms/step - loss: 1.3471 - accuracy: 0.6
309 - val_loss: 1.0296 - val_accuracy: 0.6280
Epoch 122/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0597 - accuracy: 0.6
577 - val_loss: 3.5741 - val_accuracy: 0.4978
Epoch 123/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1674 - accuracy: 0.6
612 - val_loss: 0.7116 - val_accuracy: 0.7145
Epoch 124/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0216 - accuracy: 0.6
543 - val_loss: 1.0183 - val_accuracy: 0.6271
Epoch 125/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0994 - accuracy: 0.6
528 - val_loss: 4.2866 - val_accuracy: 0.5103
Epoch 126/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1133 - accuracy: 0.6
579 - val_loss: 0.7589 - val_accuracy: 0.7056
Epoch 127/150
141/141 [==============================] - 0s 2ms/step - loss: 0.8023 - accuracy: 0.6
907 - val_loss: 4.2417 - val_accuracy: 0.4924
Epoch 128/150
141/141 [==============================] - 0s 2ms/step - loss: 1.4289 - accuracy: 0.6
135 - val_loss: 0.8191 - val_accuracy: 0.6806
Epoch 129/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0830 - accuracy: 0.6
590 - val_loss: 4.5657 - val_accuracy: 0.4924
Epoch 130/150
141/141 [==============================] - 0s 2ms/step - loss: 1.8143 - accuracy: 0.6
202 - val_loss: 6.6156 - val_accuracy: 0.4933
Epoch 131/150
141/141 [==============================] - 0s 2ms/step - loss: 1.3078 - accuracy: 0.6
579 - val_loss: 1.1882 - val_accuracy: 0.6262
Epoch 132/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0093 - accuracy: 0.6
608 - val_loss: 0.8974 - val_accuracy: 0.6646
Epoch 133/150
141/141 [==============================] - 0s 2ms/step - loss: 0.9043 - accuracy: 0.6
659 - val_loss: 0.7159 - val_accuracy: 0.7154
Epoch 134/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1211 - accuracy: 0.6
565 - val_loss: 1.0732 - val_accuracy: 0.6494
Epoch 135/150
141/141 [==============================] - 0s 2ms/step - loss: 0.8034 - accuracy: 0.6
873 - val_loss: 2.0884 - val_accuracy: 0.5227
Epoch 136/150
141/141 [==============================] - 0s 2ms/step - loss: 0.9494 - accuracy: 0.6
677 - val_loss: 2.5170 - val_accuracy: 0.5245
Epoch 137/150
141/141 [==============================] - 0s 2ms/step - loss: 0.9850 - accuracy: 0.6
516 - val_loss: 0.9857 - val_accuracy: 0.6200
Epoch 138/150
141/141 [==============================] - 0s 2ms/step - loss: 0.8984 - accuracy: 0.6
659 - val_loss: 2.7894 - val_accuracy: 0.4960
Epoch 139/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0798 - accuracy: 0.6
470 - val_loss: 1.4045 - val_accuracy: 0.5718
Epoch 140/150
141/141 [==============================] - 0s 2ms/step - loss: 0.9371 - accuracy: 0.6
610 - val_loss: 0.7155 - val_accuracy: 0.6949
```

```
Epoch 141/150
141/141 [==============================] - 0s 2ms/step - loss: 0.8770 - accuracy: 0.6
708 - val_loss: 0.6869 - val_accuracy: 0.6842
Epoch 142/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0659 - accuracy: 0.6
490 - val_loss: 0.6328 - val_accuracy: 0.7270
Epoch 143/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1052 - accuracy: 0.6
449 - val_loss: 1.9797 - val_accuracy: 0.5397
Epoch 144/150
141/141 [==============================] - 0s 2ms/step - loss: 1.0626 - accuracy: 0.6
612 - val_loss: 3.6824 - val_accuracy: 0.4951
Epoch 145/150
141/141 [==============================] - 0s 2ms/step - loss: 1.3748 - accuracy: 0.6
385 - val_loss: 1.8761 - val_accuracy: 0.5442
Epoch 146/150
141/141 [==============================] - 0s 2ms/step - loss: 0.9435 - accuracy: 0.6
724 - val_loss: 3.0516 - val_accuracy: 0.5156
Epoch 147/150
141/141 [==============================] - 0s 2ms/step - loss: 1.1698 - accuracy: 0.6
565 - val_loss: 1.1354 - val_accuracy: 0.5977
Epoch 148/150
141/141 [==============================] - 0s 2ms/step - loss: 1.3452 - accuracy: 0.6
159 - val_loss: 0.8820 - val_accuracy: 0.6833
Epoch 149/150
141/141 [==============================] - 0s 3ms/step - loss: 0.8283 - accuracy: 0.6
882 - val_loss: 6.7674 - val_accuracy: 0.4933
Epoch 150/150
141/141 [==============================] - 0s 2ms/step - loss: 1.4552 - accuracy: 0.6
322 - val_loss: 0.9250 - val_accuracy: 0.6494
```

Lets make some predictions

Unseen matchups

TeamA	TeamB
1222	1274
1462	1400
1116	1163
1211	1417
1274	1400
1163	1211
1274	1163

A CSV with the columns listed above is how we are able to feed in all matchups from the March Madness tournament.

#1 Seed Houston Vs #16 Seed North Ky.

1. Result should be above .5

In [8]:
```
# Make predictions on unseen data, Houston Vs North Ky. 1 vs 16, should be 1
unseen_data = np.array([[31, 2, 2635, 2220, 1948, 966, 611, 228, 645, 475, 226, 153, 2
prediction = model.predict(unseen_data)

# Print the prediction
print(prediction)
```

```
1/1 [==============================] - 0s 92ms/step
[[0.81716424]]
```

#8 Seed Iowa Vs #9 Auburn

1. Result should be below .5 due to the fact that Auburn upset Iowa

In [9]:
```
# Make predictions on unseen data Iowa vs Auburn 8 vs 9, should be 0
unseen_data = np.array([[19, 13, 2298, 2210, 1835, 810, 645, 214, 648, 464, 229, 86, 2
prediction = model.predict(unseen_data)

# Print the prediction
print(prediction)
```

```
1/1 [==============================] - 0s 25ms/step
[[0.05881024]]
```

#4 Seed Uconn Vs #5 SD ST.

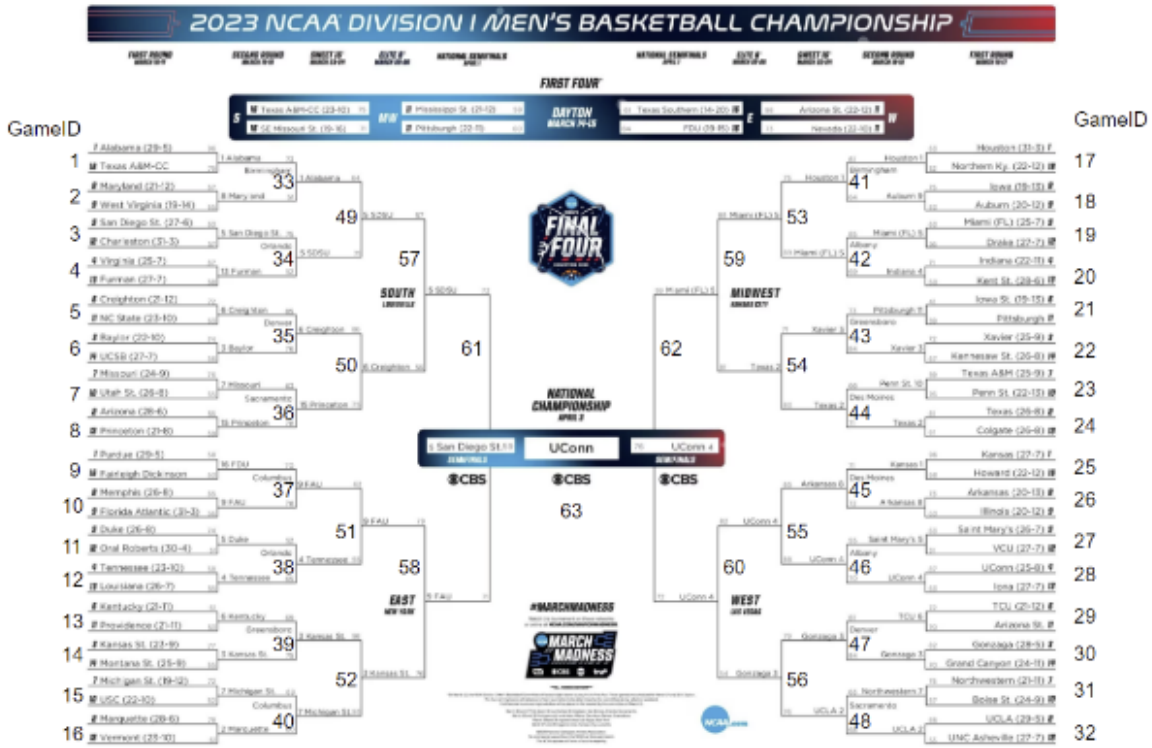
1. Result should be above .5

In [10]:
```
# Make predictions on unseen data uconn vs SD st, 4 vs 5,should be 1
unseen_data = np.array([[25, 8, 2291, 2121, 1817, 781, 667, 245, 654, 484, 211, 95, 26
prediction = model.predict(unseen_data)

# Print the prediction
print(prediction)
```

```
1/1 [==============================] - 0s 23ms/step
[[0.0073525]]
```

Inputting Multiple Matchups From One CSV

The 2023 NCAA Division I Mens Basketball Championship



In [11]:
```
# Load the unseen data
unseen_data = pd.read_csv("C:/Users/josep/Downloads/UNSEEN_TEST.csv")

# Merge the unseen data with the Perfect2023 data
unseen_data = unseen_data.merge(perfect, left_on='TeamA', right_on='TeamID').merge(per
# Extract the input features
X_unseen = unseen_data[['TotalWins_A', 'TotalLosses_A', 'TotalPF_A', 'TotalPA_A', 'Tot

# Make predictions on the unseen data
prediction = model.predict(X_unseen)

# Print the prediction
#print(prediction)

game_ids = unseen_data['GameID'].tolist()
expected_results = unseen_data['outcome'].tolist()

correct_count = 0
incorrect_count = 0


# Print the predictions along with GameID and Expected result
for i in range(len(prediction)):
    if expected_results[i] == 1:
        if prediction[i] >= 0.5:
            correct_count += 1
            print(f"GameID: {game_ids[i]}, Expected Result: {expected_results[i]}, Pre
        else:
            incorrect_count += 1
            print(f"GameID: {game_ids[i]}, Expected Result: {expected_results[i]}, Pre
    elif expected_results[i] == 0:
        if prediction[i] < 0.5:
            correct_count += 1
            print(f"GameID: {game_ids[i]}, Expected Result: {expected_results[i]}, Pre
        else:
```

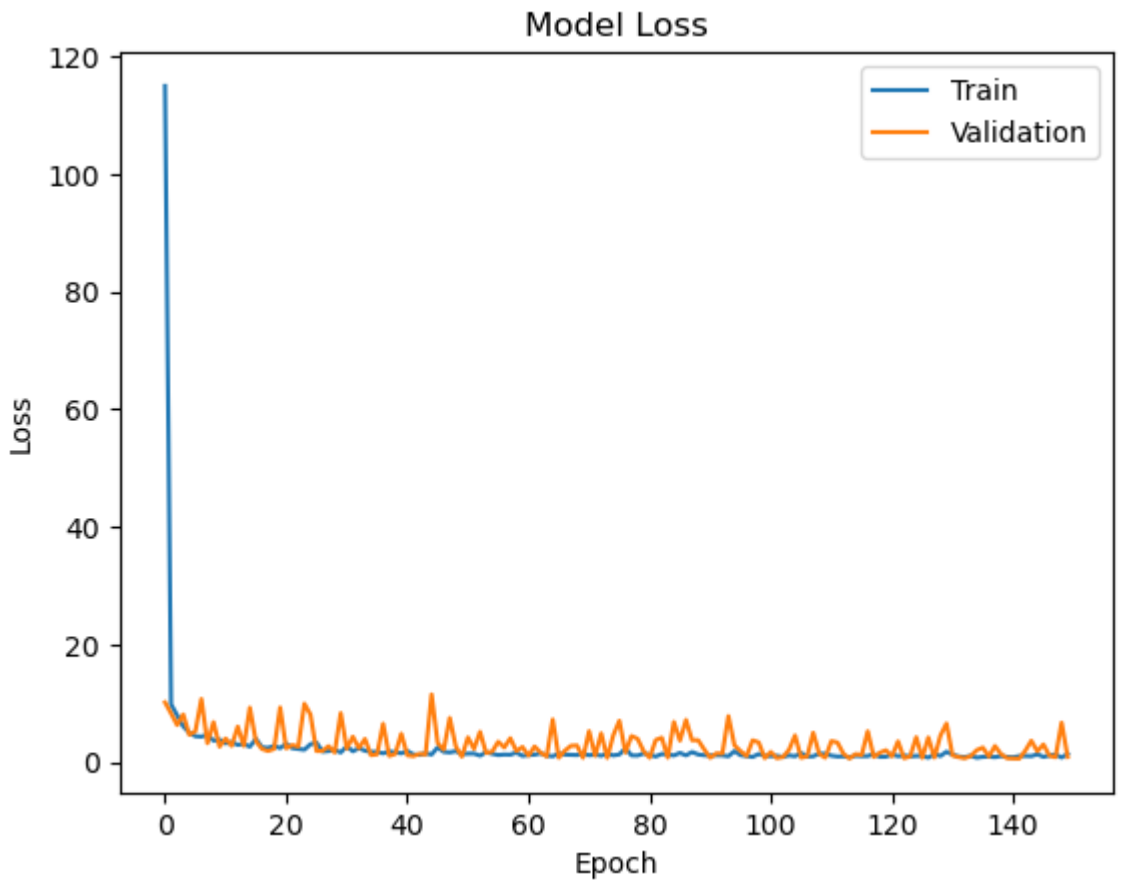
```
            incorrect_count += 1
            print(f"GameID: {game_ids[i]}, Expected Result: {expected_results[i]}, Pre

# Print the counts
print(f"\nNumber of Correct Predictions: {correct_count}")
print(f"Number of Incorrect Predictions: {incorrect_count}")

# Print the accuracy
accuracy = correct_count / (correct_count + incorrect_count)
print(f"Accuracy: {accuracy:.2%}")
```

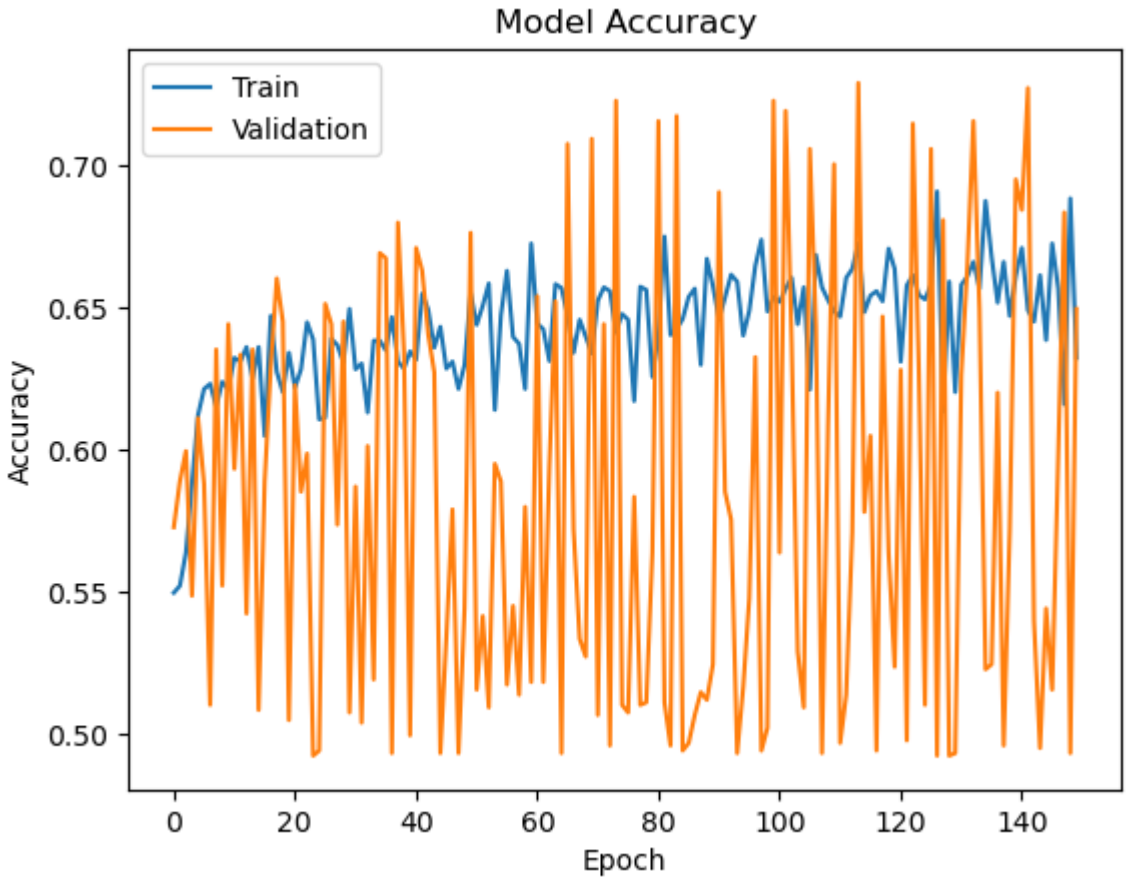
```
2/2 [==============================] - 0s 3ms/step
GameID: 1, Expected Result: 1, Predicted Outcome: 0.14 (INCORRECT)
GameID: 33, Expected Result: 1, Predicted Outcome: 0.68 (CORRECT)
GameID: 49, Expected Result: 0, Predicted Outcome: 0.07 (CORRECT)
GameID: 2, Expected Result: 1, Predicted Outcome: 0.25 (INCORRECT)
GameID: 3, Expected Result: 1, Predicted Outcome: 0.99 (CORRECT)
GameID: 34, Expected Result: 1, Predicted Outcome: 0.08 (INCORRECT)
GameID: 4, Expected Result: 0, Predicted Outcome: 0.06 (CORRECT)
GameID: 57, Expected Result: 1, Predicted Outcome: 0.34 (INCORRECT)
GameID: 61, Expected Result: 1, Predicted Outcome: 0.01 (INCORRECT)
GameID: 10, Expected Result: 0, Predicted Outcome: 0.01 (CORRECT)
GameID: 37, Expected Result: 0, Predicted Outcome: 0.00 (CORRECT)
GameID: 63, Expected Result: 0, Predicted Outcome: 0.04 (CORRECT)
GameID: 62, Expected Result: 0, Predicted Outcome: 0.04 (CORRECT)
GameID: 55, Expected Result: 0, Predicted Outcome: 0.01 (CORRECT)
GameID: 46, Expected Result: 0, Predicted Outcome: 0.04 (CORRECT)
GameID: 5, Expected Result: 1, Predicted Outcome: 0.03 (INCORRECT)
GameID: 35, Expected Result: 1, Predicted Outcome: 0.05 (INCORRECT)
GameID: 50, Expected Result: 1, Predicted Outcome: 0.26 (INCORRECT)
GameID: 8, Expected Result: 0, Predicted Outcome: 0.73 (INCORRECT)
GameID: 36, Expected Result: 0, Predicted Outcome: 0.48 (CORRECT)
GameID: 6, Expected Result: 1, Predicted Outcome: 0.01 (INCORRECT)
GameID: 7, Expected Result: 1, Predicted Outcome: 0.07 (INCORRECT)
GameID: 9, Expected Result: 0, Predicted Outcome: 0.13 (CORRECT)
GameID: 31, Expected Result: 1, Predicted Outcome: 0.02 (INCORRECT)
GameID: 11, Expected Result: 1, Predicted Outcome: 0.15 (INCORRECT)
GameID: 38, Expected Result: 0, Predicted Outcome: 0.20 (CORRECT)
GameID: 51, Expected Result: 1, Predicted Outcome: 0.18 (INCORRECT)
GameID: 12, Expected Result: 1, Predicted Outcome: 0.06 (INCORRECT)
GameID: 13, Expected Result: 1, Predicted Outcome: 0.07 (INCORRECT)
GameID: 39, Expected Result: 0, Predicted Outcome: 0.06 (CORRECT)
GameID: 58, Expected Result: 1, Predicted Outcome: 0.31 (INCORRECT)
GameID: 14, Expected Result: 1, Predicted Outcome: 0.10 (INCORRECT)
GameID: 52, Expected Result: 1, Predicted Outcome: 0.17 (INCORRECT)
GameID: 15, Expected Result: 1, Predicted Outcome: 0.01 (INCORRECT)
GameID: 40, Expected Result: 1, Predicted Outcome: 0.00 (INCORRECT)
GameID: 16, Expected Result: 1, Predicted Outcome: 0.44 (INCORRECT)
GameID: 17, Expected Result: 1, Predicted Outcome: 0.82 (CORRECT)
GameID: 41, Expected Result: 1, Predicted Outcome: 0.86 (CORRECT)
GameID: 18, Expected Result: 0, Predicted Outcome: 0.06 (CORRECT)
GameID: 53, Expected Result: 0, Predicted Outcome: 0.46 (CORRECT)
GameID: 19, Expected Result: 1, Predicted Outcome: 0.04 (INCORRECT)
GameID: 42, Expected Result: 1, Predicted Outcome: 0.16 (INCORRECT)
GameID: 59, Expected Result: 1, Predicted Outcome: 0.08 (INCORRECT)
GameID: 54, Expected Result: 0, Predicted Outcome: 0.97 (INCORRECT)
GameID: 44, Expected Result: 0, Predicted Outcome: 0.01 (CORRECT)
GameID: 20, Expected Result: 1, Predicted Outcome: 0.00 (INCORRECT)
GameID: 21, Expected Result: 0, Predicted Outcome: 0.03 (CORRECT)
GameID: 22, Expected Result: 1, Predicted Outcome: 0.73 (CORRECT)
GameID: 23, Expected Result: 0, Predicted Outcome: 0.24 (CORRECT)
GameID: 24, Expected Result: 1, Predicted Outcome: 0.05 (INCORRECT)
GameID: 25, Expected Result: 1, Predicted Outcome: 0.36 (INCORRECT)
GameID: 45, Expected Result: 0, Predicted Outcome: 0.35 (CORRECT)
GameID: 26, Expected Result: 1, Predicted Outcome: 0.15 (INCORRECT)
GameID: 27, Expected Result: 1, Predicted Outcome: 0.02 (INCORRECT)
GameID: 28, Expected Result: 1, Predicted Outcome: 0.01 (INCORRECT)
GameID: 60, Expected Result: 1, Predicted Outcome: 0.01 (INCORRECT)
GameID: 47, Expected Result: 0, Predicted Outcome: 0.00 (CORRECT)
GameID: 29, Expected Result: 1, Predicted Outcome: 0.02 (INCORRECT)
GameID: 30, Expected Result: 1, Predicted Outcome: 0.56 (CORRECT)
```

```
GameID: 56, Expected Result: 1, Predicted Outcome: 0.01 (INCORRECT)
GameID: 48, Expected Result: 0, Predicted Outcome: 0.00 (CORRECT)
GameID: 32, Expected Result: 1, Predicted Outcome: 0.11 (INCORRECT)
GameID: 43, Expected Result: 0, Predicted Outcome: 0.16 (CORRECT)

Number of Correct Predictions: 27
Number of Incorrect Predictions: 36
Accuracy: 42.86%
```

In [12]:

```python
import matplotlib.pyplot as plt

# plot the training and validation loss over epochs
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.show()

# plot the training and validation accuracy over epochs
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```





Results

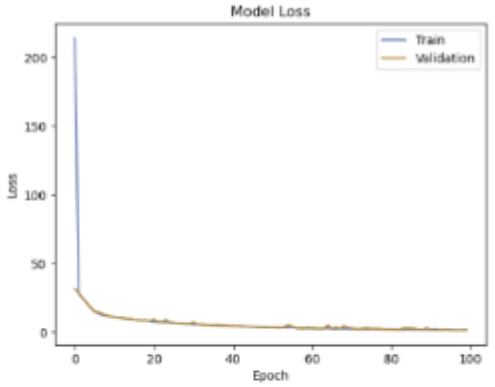
Single Layer Neural Network

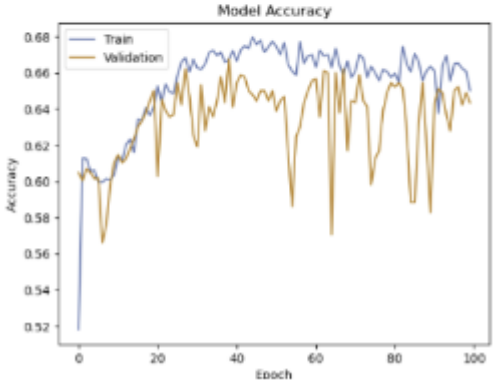
1. Single Layer Neural Network

Clumped 1/0 input data:

a. Accuracy: ~66.5%

b. This was a single layer neural network. One input layer, 24 nodes + bias term, as well as one output layer, 1 node. The activation function was sigmoid. This model was likely to over fit.





New Random 1/0 input data:

c. Accuracy: ~65%

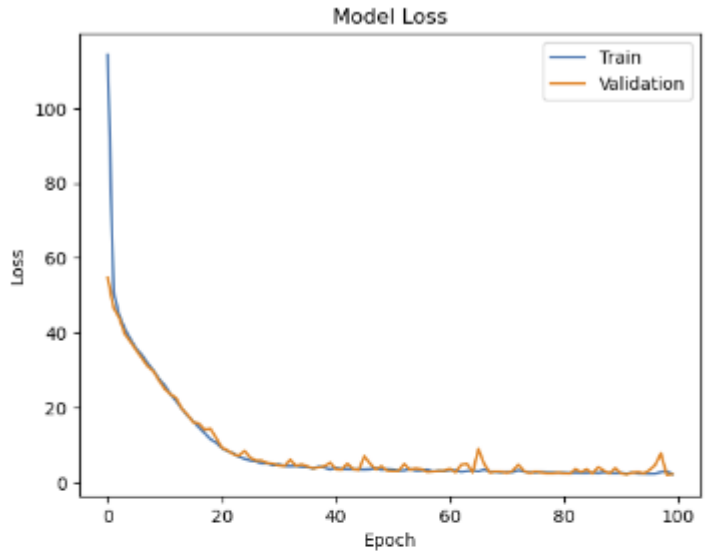
d. The change of making the data more random in terms of placement of winning and losing inputs did not change the performance of the NN too much but it did decrease the accuracy a little which is to be expected with less perfect data than what was inputted before.

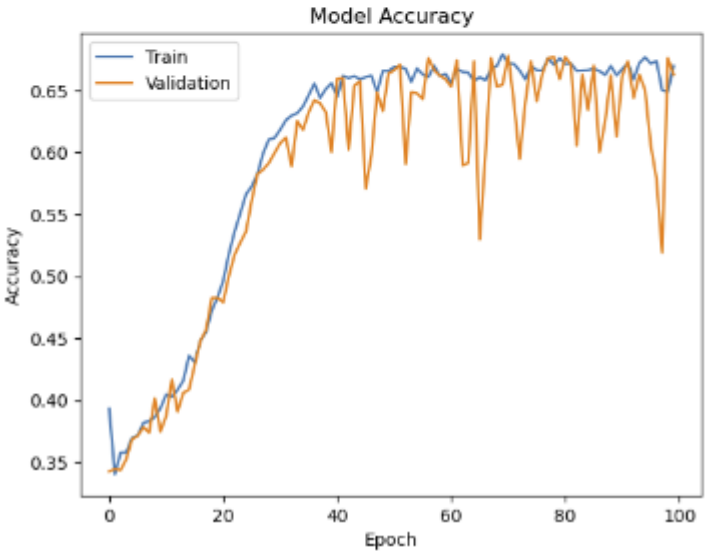
e. This model performed great when tested on the whole tournament! Here are the results:
Number of Correct Predictions: 42

Number of Incorrect Predictions: 21

Accuracy: 66.67%

66% Accuracy is AWESOME!





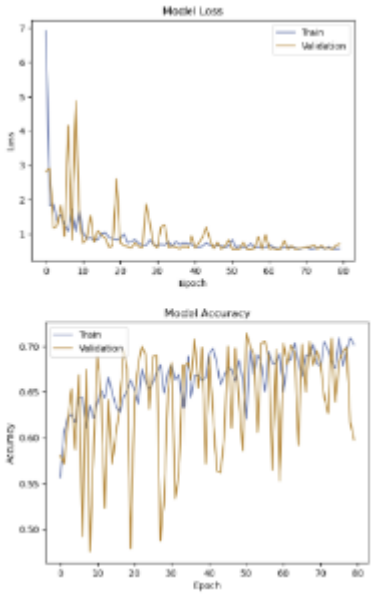
Multi layer Neural Network

1. Multi-layer Neural Network

Clumped 1/0 input data:

a. Clumped 1/0 input data: Accuracy: ~65-70%

b. This was a multi-layer neural network. One input layer, 24 nodes + bias term, 1 hidden layer also with 24 nodes, with the actication function being relu, as well as one output layer, 1 node. The activation function was sigmoid for the output layer. This model was highly likely to over fit.



New Random 1/0 input data:

c. Accuracy: ~65%

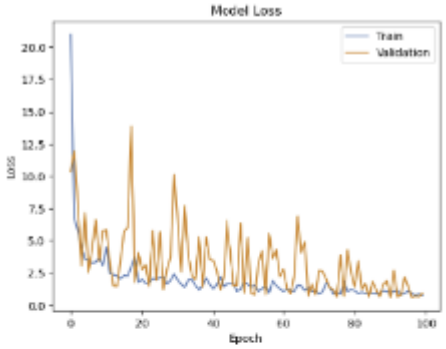
d. The change of making the data more random in terms of placement of winning and losing inputs did not change the performance of the NN too much but it did decrease the accuracy a little which is to be expected with less perfect data than what was inputted before.

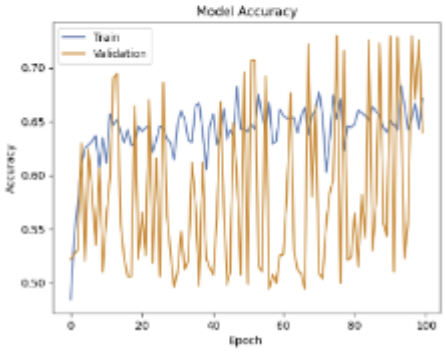
e. This model performed very poorly when tested on the whole tournament. The results were as follows:

Number of Correct Predictions: 26

Number of Incorrect Predictions: 37

Accuracy: 41.27%





XGBoost

1. XGboost

   a. Accuracy: 89.22%

   b. This was an XGBoost model. We have not yet looked deep enough into this to make sure that it is accurate. However if this accuracy is correct it will be outpreforming the neural networks by 20% which is a massive step up and much closer to perfect.

   c. XGBoost did perform quite well on the whole tournament data. Here are the results:

   Number of Correct Predictions: 38

   Number of Incorrect Predictions: 25

   Accuracy: 60.32%

Conclusion

In conclusion we found that the best results came with the use of a single-layer neural network. The accuracy of the single-layer neural network, for the training data, hovered right around .65. This is about 15% higher than that of a random guess. This model took many steps to get to. The first set of input data was poorly implemented so therefore the model was highly overfit and would consistently guess a lower value. With the new input data there is a slight drop in accuracy but a much better fit of the data. Though the highest accuracy we received on the test

data was 66% from the single-layer NN the most consistent accuracy came with that of XGBoost. It will always give a result in the 60% range when given new test data. We did run out of time on this project in order to research this more but it seems to be the best possible route to take. The next steps in this project include increasing the accuracy of the XGboost model and researching the best ways that XGboost can be used to help us in this project.

Interesting point: the graph showing validation accuracy has an out of control up and down pattern in both the single and multi-layer NN, Wondering why this is?

Based on the XGboost model the winner of the tournament would be: Purdue. The final four would include: Arizona vs. Purdue & Texas A&M vs. Iona

Works Cited

"March Machine Learning Mania 2023." Kaggle,

https://www.kaggle.com/competitions/march-machine-learning-mania-2023/overview.