Predicting March Madness using Neural Networks

Benjamin Brookhart & Nathan Prentiss
Dept. of Computer Science
University of Massachusetts
Lowell, MA 01854, USA
Benjamin_Brookhart@student.uml.edu Nathan_Prentiss@student.uml.edu

***Abstract*-- The NCAA Men's Basketball Tournament is an established yearly tournament with crazy upsets, drama and excitement. Many people enjoy trying to pick the winners, whether they have a lot of basketball knowledge or not. Our goal is to combine two datasets – one being dozens of metrics covering all facets of basketball, and the other being the results of those teams in the tournament to train a model to predict the results of teams it hasn't seen yet. Implementing a successful model could prove to be valuable to sports betting companies, as having informed predictions prevents people from being able to win money off them consistently. Our model is an attempt to solve the seven-class problem of tournament wins, and gets the exact number of wins correct nearly half of the time.**

Introduction

March Madness is an annual highly anticipated sports tournament featuring the top 64 NCAA D1 basketball teams. Our study will be focused only on the men's bracket. People all over the country make their own brackets trying to as perfectly as possible predict the results of the tournament. March Madness is statistically the highest sports betting event in America. This year, it's estimated $3.1 billion will be legally bet on this event. The most accurate bracket happened in 2019 where a man predicted the first 49 results correctly, which is a little more than the entire first two rounds correctly. On average, the favored team, being the higher seeded team wins 70% of the time. In this project, we will be using a pre-made dataset containing advanced metrics like offensive and defensive ratings, efficiencies and other data to build a model that can determine what leads to winning basketball during March Madness. Our goal is to be able to train a model that can predict how many wins a team got in that tournament given metrics alone.

Related Works

Gumm, Barrett, and Hu used an ensemble regression technique which gave them a Log Loss of 0.56411, which proves to be better than a base lower-seed strategy, which got a Log Loss of 0.60021. https://ieeexplore.ieee.org/abstract/document/7176206

Evan also used a basic neural network technique with the same mae and mean squared loss functions we did to try and predict the winner of each game as compared to us predicting how many wins a team got a tournament, with an accuracy of about 55%. https://www.kaggle.com/code/ironicninja/simple-nn-for-march-madness-predictions

Description

Database Details:

We are using 2 Kaggle Datasets for this project, one for team data, and one for the Bracket Results. We believe this will not cause conflicts since it is the same data. The team statistics are from the "March Madness Historical DataSet (2002 to 2025)"

This dataset contains any type of information a professional basketball analyst would use to make informed decisions such as a team's offensive and defensive ratings, shooting percentages, coaching tenures, average player heights and so on. These are the numbers we train our model on to see if it can recognize a winning basketball program given their metrics for that year, which is independent of all other instances of that team. https://www.kaggle.com/datasets/jonathanpilafas/2024-march-madness-statistical-analysis/data

The bracket results from 2008 to 2024 are from the "March Madness Data" Dataset. This includes the team information, their seed that year, and a number that implies the round they made it to in the tournament. We use this information to

calculate the amount of wins each team had in the tournament which is done in preprocessing.

https://www.kaggle.com/datasets/nishaanamin/march-madness-data?select=Tournament+Matchups.csv

Preprocessing:

We started by manually changing some of the data from both datasets to align with each other. For example, some of the team name abbreviations would be represented differently so we would find all instances of this and make the names of the teams consistent. There were also cases where the datasets conflicted in certain teams that made the March Madness tournament. In these instances we checked the official brackets online and made the appropriate changes to the datasets.

We dropped rows of teams in datasets of data we would not process. Years 2002-2007 were dropped from the metrics dataset because the results dataset did not have information regarding these years. Year 2020 was dropped because the tournament was cancelled due to the COVID outbreak, and 2025 had not yet happened yet so there were no results either.

Columns of information that would not help train our model or had incomplete data that could not be manually put in by us were also dropped. Those columns include Coach Name, Full Team Name, NST rates, among others.

Teams that did not make the March Madness tournament that year were dropped, for example UMass Lowell's team last year. We created our own metric based off of the results data that simplified things into how many wins a tournament team got that year ranging from 0-6. 0 being a team lost in the first round, 6 being they won the tournament. This can be done given all the teams we are training the model on played in at least 1 tournament game.

We parsed the results data and used a column indicating which placement they got (1, 2, 4, 8, 16, 32, 64), and used a formula of

$$wins = 6 - log_2(placement)$$

to convert that to the number of wins.

To add the column of number of wins in the metrics dataset, we had to use a dictionary mapping the combination of year and team with the number of wins they got.

Algorithms, Training, Testing:

We took our preprocessed data and stored it in value x. The data given gets normalized using the formula (value - mean) / standard deviation. Our t value is an array of values 0-6 that represent the amount of wins an "x" team won in the tournament. This data gets train test split using our own function. We do this so the model can train using unordered data. The indexes of x are randomly shuffled using the numpy shuffle function and split by our trained percentage of .8. The values of x_train, x_test, t_train, and t_test are now set and split with values of x and t then returned.

This newly trained data is then smoted using the SMOTE (Synthetic Minority Oversampling Technique) function. This algorithm allows us to give some balancing to the dataset given we have 32 times more teams that won 0 games than championship winners. (32 teams get first rounded every year and only 1 team wins the championship). Otherwise the dataset would have significantly less successful teams to sample on and predict 0 wins for the majority of teams.

After the data is smoted, we now have a new representation of target values. We implement the one-hot encoding technique to store these final values into a list of arrays of size 7. These arrays have a "1" at the position of the amount of wins a team got and a "0" in all others. For example, if a team won 4 games, the one-hot array for that specific piece of data would look like [0,0,0,0,1,0,0].

X_train_smoted holds the random amount of total samples added to the data plus the original data as well. These are reshaped into 3d arrays of size [total_samples, 1, columns of metrics] to properly generate weights in that upcoming function.

We generate two weights. The first being the weights of the data in our columns (128) to the number of neurons in our hidden layer (300). The latter being the number of neurons in the hidden layers (300) to our output layer of target values of 7 possible outcomes. The generate weight function gives us a 2d array of random values the size of the dimensions of the amount of connections in our network.

Using these weights we can now weigh them on our final fitted data and teach the model to calculate metrics of loss and accuracy. We use a grid search method to determine the best number of

layers, learning rate, and interactions. This tests our model on different combinations of parameters among those three. The train function is where we go through the stages of neural networks, forward propagation, loss, and backward propagation. Our propagations use only 1 hidden layer, we do dot product and sigmoid operations on the x values. Our loss function uses three different functions to improve the training. We use Root-Mean Squared Error, Mean Absolute Error, and R-Squared Loss functions. These values are added together to give us our loss at each iteration of x. The weights then get updated using back-propagation to minimize our given errors.

We call our own accuracy scoring function using the values of forward propagated x and target values t at index i. This function has two different forms of scoring. The first is a basic if the prediction is the same as the actual, then the model makes a correct, accurate decision and it will be scored in the accuracy. Otherwise, the model was incorrect and no credit is given. The second way is a weighted score that rewards guessed made by the model that were 1 or 2 games off the target. For example, the model predicting a team that won 6 games to win 4 is much different than predicting that same 6 win team to win 0 games. Therefore now, if a team is guessed correctly it is given a score of 1, one off awarded .5, two off rewarded .25 and anything else is unrewarded 0.

Our training function does very well and gets nearly every classification correct by the end of the 100 iterations, the combined loss is under 0.04.

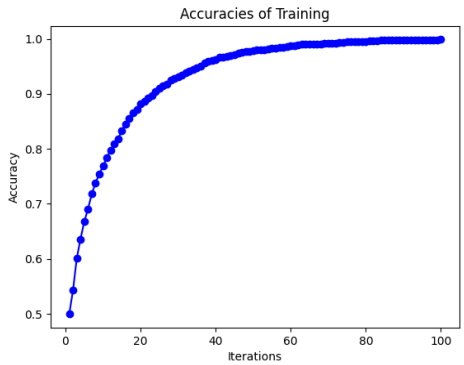

Figure 1.1: Raw Accuracy of Training Set Over 100 Iterations

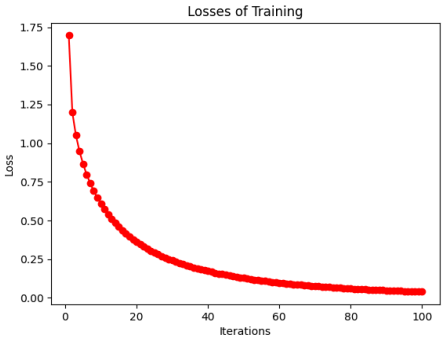

Figure 1.2: Combined Loss of Training Set Over 100 Iterations

Back in the train function, the value “right” adds a 0 or 1 for the first case of completely right or not and “weighted” adds the calculated weighted value to the total. These totals are then divided by the total samples to generate an accuracy score that represents how many teams the model was able to accurately predict won x amount of games. These accuracies and losses are for our trained model.

Given these accuracies and losses at each training point, we do the same training processes for the test data. Forward propagation, loss and the is_right functions are used to calculate accuracies of the test data in the same way. The new accuracy is updated to represent the test values and these values are our official raw and weighed accuracies.

The data the model predicted was evaluated using a Confusion Matrix (See Figure 1.3 below). The rows represent the actual number of wins and the columns represent the predicted number of wins.

Experimental Evaluation

Libraries:

We used the Pandas library to read and write from CSV files, and used the DataFrame functionality of it to add, remove, and modify columns and rows. NumPy was also used to generate random numbers, perform matrix operations, store data in NumPy arrays, and calculate other mathematical operations.

Since half of the teams get eliminated each round, we had about half of our training set as teams in the zero wins class, about a quarter with one win, and so on. To mitigate this, we used SMOTE to balance the classes so that each of the seven classes had equal data points for the training set. We used the

SMOTE class from imblearn.over_sampling to do this.

To get the results of our model, we used the confusion_matrix feature from the sklearn.metrics library and used matplotlib to create visuals for these.

Starter Code:

We used the geeksforgeeks implementation of a neural network as a starter code, using the basic implementation of forward propagation and backpropagation. https://www.geeksforgeeks.org/implementation-of-neural-network-from-scratch-using-numpy/

Results:

In our different trials, we were able to achieve a raw accuracy of 43 to 50 percent with a combined loss function of about 1.19 to 1.35. The weighted scores ranged from around 60 to 70 percent. We believe this is reasonably good since we are not predicting each game, but rather how many wins a team will get specifically, which can be very difficult to do. Generally, teams that win the tournament are comparable to teams that have won a few games, so predicting exactly which amount of wins a team will fall under is quite a task.

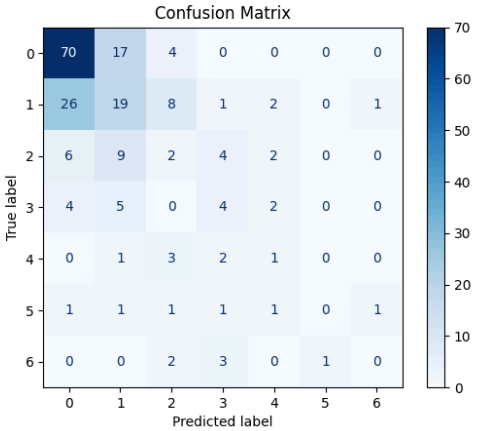

Figure 1.3: Confusion Matrix of Tested Model

From Figure 1.3 we can see that the model generally picked up on which teams made a deep run and not, sometimes getting one game off with tournament winners. There are outliers, which is reasonable given that usually there is a "juggernaut" that goes out early or a "Cinderella" team that greatly exceeds expectations.

Limitations

One of the limitations that we could have is the number of layers in our network. Our starter code had a single hidden layer, along with the input and output layers, and if we had implemented more layers to the network, it may have given better results.

We could also have implemented this as a predictor instead of a classifier, which likely would have gotten us more comparable results with the other people who have attempted a solution for the March Madness problem. One could implement this classification model as a sort of predictor, by comparing the teams that played and seeing if one was in a class of more expected wins or not. If there was a tie, we could use the pre softmax output instead of the maximum prediction's class.

If we wanted to simplify the problem, we could have classified the teams differently. For example, instead of a seven class problem, we could have used only three or four classes such as, "0 wins", "1 win", "2-3 wins", and "4-6 wins". This could have simplified the problem for the model and boosted the scores.

While there will almost always be a discrepancy between training and testing results, the large difference could be attributed to overfitting, which could be resolved using a variety of techniques.

Conclusion

Our process of preprocessing, training and testing yielded solid results showing the effectiveness of a one-hidden layer neural network. The seven-class problem is a difficult one, especially with imbalanced classes and the general chaos that occurs during the tournament. Generally, the loss of the first iteration of training was around 1.6 to 1.7. If you compare this to the loss of the test set, the test loss is lower, proving that the model did learn a bit about what allows a basketball team to win.

The March Madness NCAA Tournament has been a staple in American sports culture, holding tournaments since 1939. Predicting the games can be a fun challenge for a casual sports fan as even the best analysts can pick wrong. There is even a massive sports betting industry nowadays that can stand to gain a lot of money by providing accurate money

lines and promotions based on good predictions. As a machine learning project, we found the metrics to be interesting and this problem to be a good challenge and we believe our model has done decently well on it.