CMPE 452
Neural and Genetic Computing

NBA Wins-Made Predictions From College Stats

Alan Dimitriev (20062431)
Adam McCaw (20071933)
Noah Rowe (20056456)
Hayden Wang (20056655)

December 12, 2020

Problem Description and Motivation

The aim of our project was to use machine learning to create a model that could accurately predict the “Wins-Made” an NBA draft prospect would have contributed to an NBA roster based off of their college basketball stats (such as points per game, assists and rebounds) and their NBA draft combine measurements (such as height, wingspan, and vertical leaping ability). The NBA is the third most profitable professional sports organization in the world, and advanced analytics is slowly becoming more prevalent in the league with teams like the Houston Rockets having their general manager Daryl Morey construct their entire roster to play to what his analytics department deemed “the most efficient brand of basketball”. The NBA draft is the only way for teams to secure fresh talent entering the league, meaning that each draft pick has immense value to the team that holds it. Due to how profitable basketball is, NBA teams are becoming more interested in investing in advanced analytical and predictive methods to help ensure that when they select a player with their draft pick, they are adding someone to the roster who will benefit the team. We all have a mutual interest in sports and were excited by the idea of applying class concepts to a subject matter that we all follow very closely. We also wanted to push the boundary of what was achievable in this project by collecting the data ourselves instead of downloading a premade dataset.

Data Description and Motivation

The data used for this project was independently scraped from multiple basketball-focused websites, including BasketballReference.com, BasketballGM.com, ESPN.com, and NBA.com. The data was collected using the commercial program OctoParse. This allowed for rapid automated web-scraping that provided the desired data in comma separated value (csv) format, which worked well with the Python packages used.

The data collected spanned from 2001 to 2018, and focused on both college and NBA level data for approximately 1300 players. Each of these players was drafted within this timespan for an NBA team. The feature set we developed included college statistics, player biometrics information, player high school rankings, and mock draft results. Table 1 shows each of the feature columns as well as what website they were obtained from. In depth descriptions of each statistic can be found at their source website.

Table 1: An outline of all the features used in this analysis along with where the data was collected from.

Collected Statistic	**Source**
Mock draft results	Draftexpress.com
Recruiting Services Consensus Index	Draftexpress.com
Number of NCAA awards	Basketball.realgm.com
Strength of schedule	Basketball.realgm.com
Position	NBA.com
Wingspan	NBA.com
Reach	NBA.com
Body fat percentage	NBA.com
Hand width	NBA.com
Hand length	NBA.com
Height	NBA.com
Weight	NBA.com
Maximum vertical leap	NBA.com
Shuttle run (sprint)	NBA.com
Lane agility	NBA.com
Bench-press repetitions	NBA.com

Age	Basketball-reference.com
Games played	Basketball-reference.com
Minutes	Basketball-reference.com
Effective field goal percentage	Basketball-reference.com
Offensive rebounding percentage	Basketball-reference.com
Defensive rebounding percentage	Basketball-reference.com
Assist percentage	Basketball-reference.com
Turnover percentage	Basketball-reference.com
Steal percentage	Basketball-reference.com
Block percentage	Basketball-reference.com
Usage percentage	Basketball-reference.com
Minutes played per foul	Basketball-reference.com
Free throws per field goal attempt	Basketball-reference.com
Minutes per three-point attempt	Basketball-reference.com
Player efficiency rating	Basketball-reference.com
Points per shot	Basketball-reference.com
Dean Oliver’s offensive rating	Basketball-reference.com
Dean Oliver’s defensive rating	Basketball-reference.com

Offensive win shares	Basketball-reference.com
Defensive win shares	Basketball-reference.com
Three-point percentage	Basketball-reference.com
Free throw percentage	Basketball-reference.com

The target variable for this analysis was the Wins Made of a player. Wins Made is designed to estimate the amount of wins throughout the season that a specific player will provide for their team. Wins Made takes into account offensive and defensive performances.

The next step was to process this data so it can be used within our models. This involved eliminating data entries that we did not wish to model. This included international players, as they did not have the college statistics we required. We also removed players that did not play a full year in the NBA. This was because they did not have the target variable we are looking to predict, meaning that they could not be used in the training or validation process. This resulted in ~700 useful entries to use within our models.

Most data columns contained missing values which had to be addressed before model training. This was done using mean imputation. Mean imputation addresses missing values by replacing them with the mean value of that column. All columns considered for this were numeric columns, so missing values were replaced by the arithmetic mean of the entire column. Another option considered to address the issue of missing values was multiple imputation by chained equations (MICE), which can be implemented in Python through sci-kit learn. MICE uses the present information in each row to estimate what the missing value is likely to be. This can be useful as it allows for more accurate imputations, and it maintains some of the initial column distribution. However, seeing as data preparation was not the focus of this assignment and MICE is fairly complicated to implement and understand, it was not used.

Lastly, each feature column had to be scaled for optimal model performance. The two scaling techniques entertained was standard scaling and min-max scaling, both of which were implemented using the Python package sci-kit learn. Standard scaling works by removing feature column mean and dividing by the standard deviation to create unit variance throughout the column. Min-max scaling maps all column values between 0 and 1, where 1 is the largest column value and 0 is the smallest. Different scaling techniques were found to be most effective across the 4 models outlined in this report, so feature scaling was chosen depending on the model.

Finally, Figure 1 shows an overview of all data preprocessing steps.

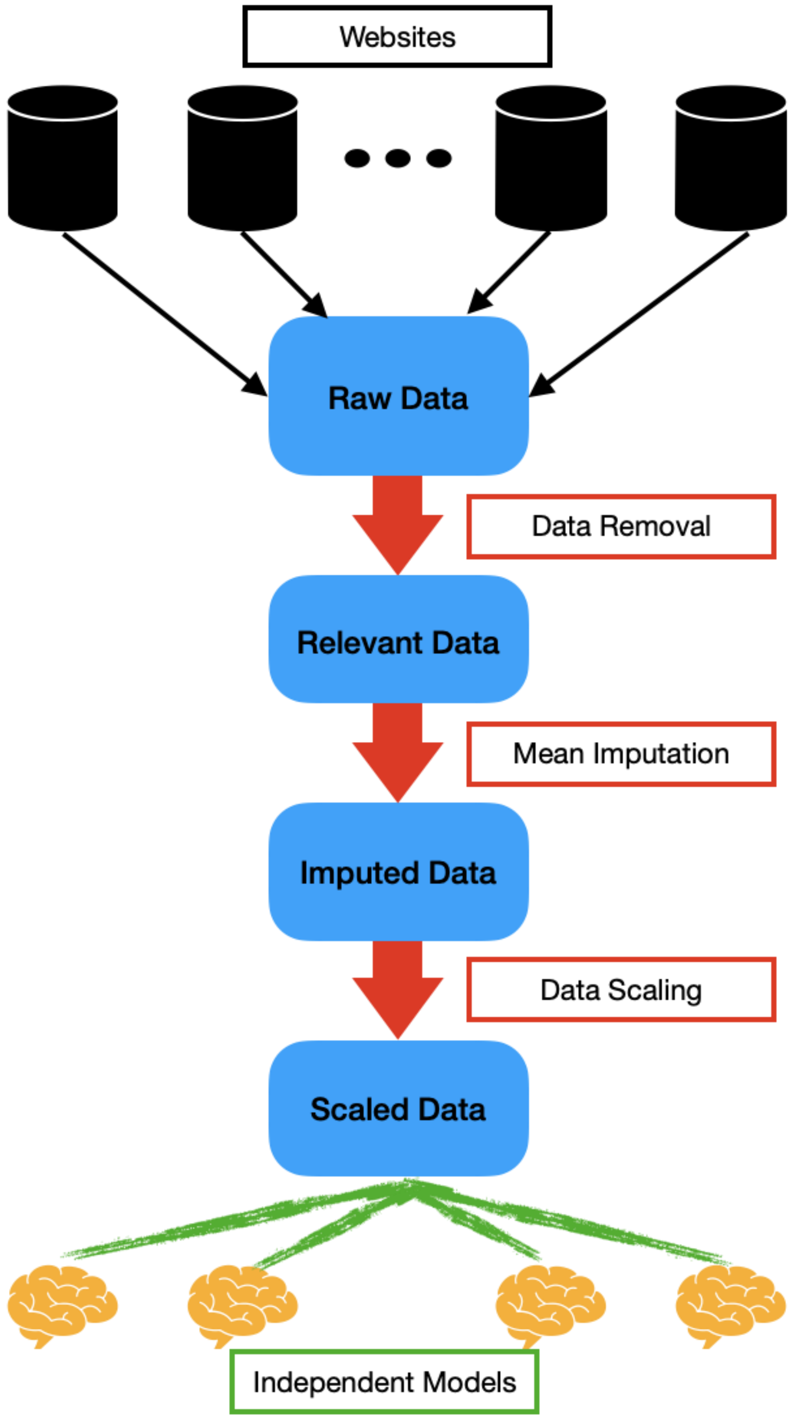

Figure 1: An outline of the data processing steps taken to create the final datasets used within each model. Data was gathered from websites using OctoParse, and all other transformations were performed in Python

Model Implementation and Validation

Radial Basis Function Network (Hayden Wang)

The model I used is an RBF network that uses a 3-layer feedforward neural network similar to a MLP with one hidden layer. The hidden layer consisted of RBF nodes which used a Gaussian activation function instead of something like Sigmoid which you would see in a standard MLP. The specific function either gaussian, multi-quadratic, or thin plate spline doesn't really affect the performance of the network as long as $f(x) = f(-x)$ and $f(x) = 0$ when x approaches infinity where f denotes the activation function. I used 530 total nodes in the hidden layer which corresponds to the number of training samples we had for our data. The reason I didn't use any techniques to reduce the number of nodes was due to the fact that we had such a small dataset that it was still feasible to have as many hidden nodes as training samples. In the case that we work with more data, it will be necessary to implement strategies for reduce the number of nodes. The weights for the hidden nodes were initialized with a uniform distribution which through trial and error seemed to be the best performing initial setup. In reality without the constraints of being required to implement a neural network, I would have chosen another model to predict the wins made, I chose a RBFN due to initial testing showing that it passed the minimal threshold of success such that it was a model that could perform decently given the distribution and structure of the data. This is in no way the best model for the project, but gives valuable insight in how RBFN's perform on data that isn't the best suited for this particular model.

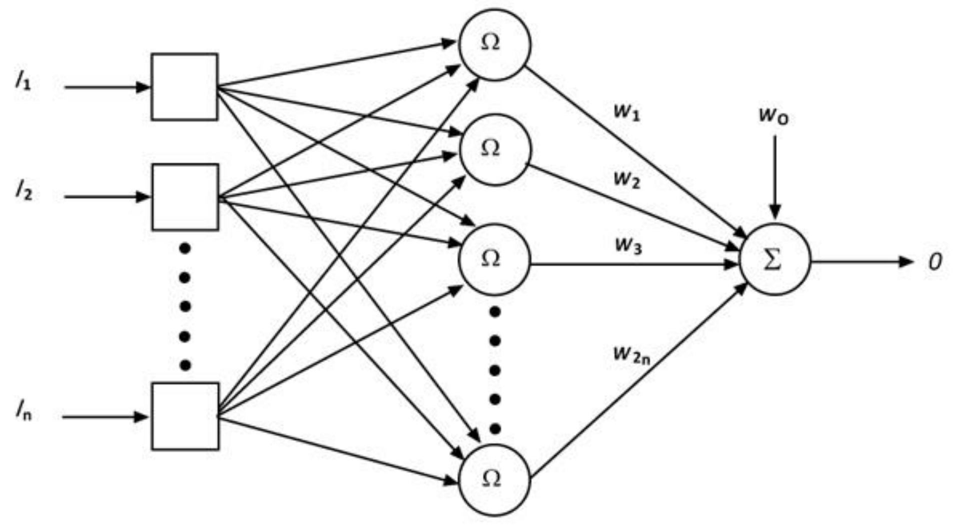

Figure 2: Graph showing the rough structure of RBFN implemented

In terms of preprocessing, what I did differently compared to the other members of my group involved a few feature reduction techniques. Starting off with all the features I manually filtered out features that would not improve the predictive power of the model such as win-share (used to calculate the target variable wins-made), player position, NBA team they played for, etc. I made sure not to filter any features based on what I personally felt would have predicative power because with these types of problems it is difficult to know straight away what a good predictor of our target variable might be. Instead, a systematic approach was taken where I first filtered out constant features which did not vary at all. This is an obvious filter because a feature which values that are the same for all samples is not helpful in any way in term of predictive power. Next, I filtered out features that varied by less than 0.01 which was found through trial and error to be a suitable number which did not reduce the feature set by too much while at the same time still doing a decent job at filtering out some features that did not vary by much. Lastly, I calculated the linear correlation between every pair of features to see if I could remove one of two pairs that correlated highly with each other. I chose an 80% correlation to be that cut-off. From this I found features such as height_with_shoes to be highly correlated with height and removed one of the two such pairs. After performing the three above mentioned feature reduction techniques I end up with 21 features total which was a better ratio for our smaller dataset of 530 training samples. Furthermore, I performed a normalization of all the features using MinMaxScaler from scikit-learn so that the varying magnitudes for each feature were now comparable and the model would not be biased towards one feature simply for inherently larger or more spread-out values.

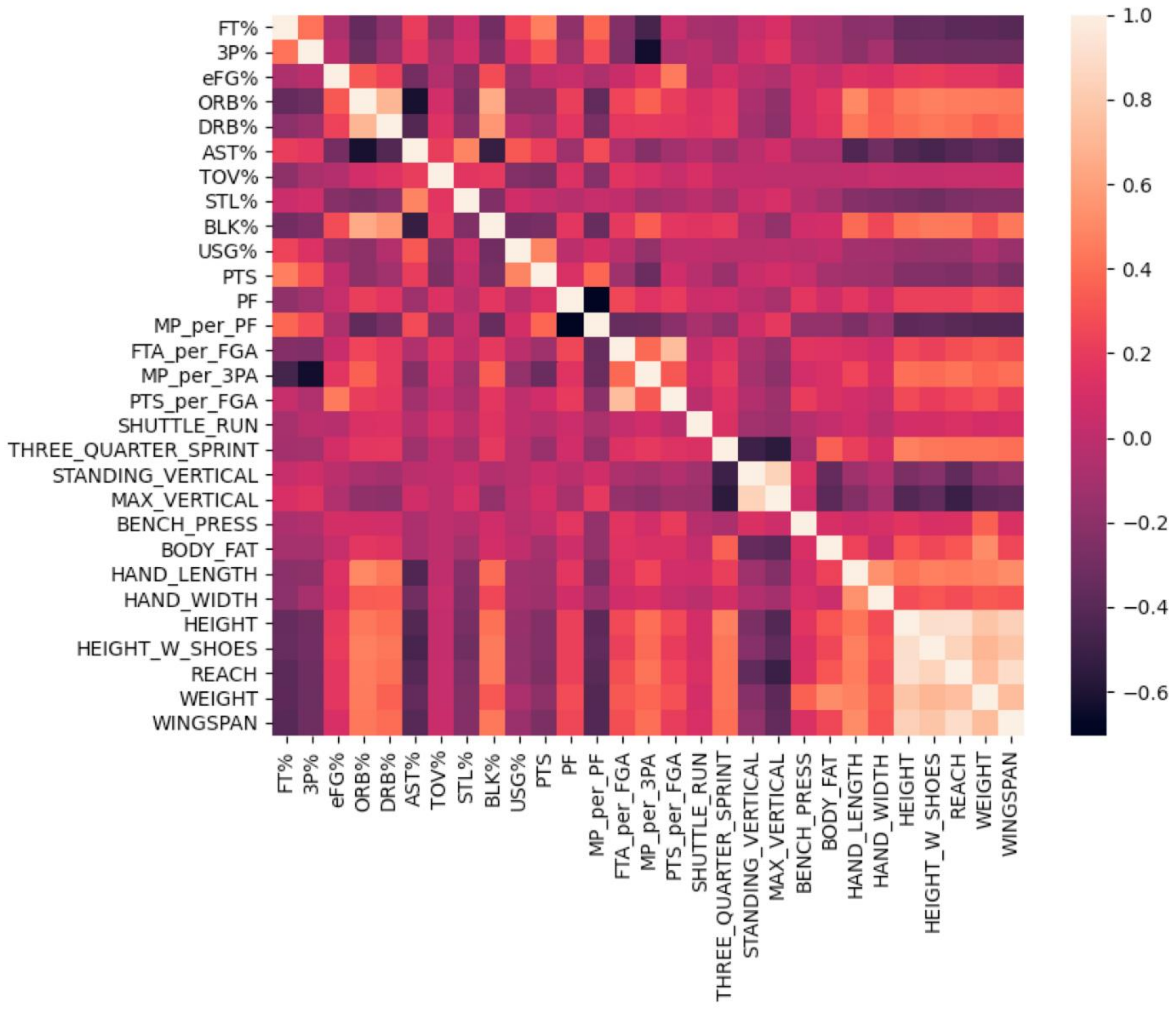

Figure 3: Heatmap showing the linear correlation between different features

Segregated Multilayer Neural Network with Dropout (Noah Rowe)

This model focused on creating three independent models that allowed for extracting unique aspects dependent on player position. The provided dataset contained players from three positions; Center, Forward, and Guard. By developing a model specific to each position, it would allow for each model to identify and learn the most impactful relationships between Wins Made and player features for each position. Figure 4 shows the segregated model layout and evaluation approach.

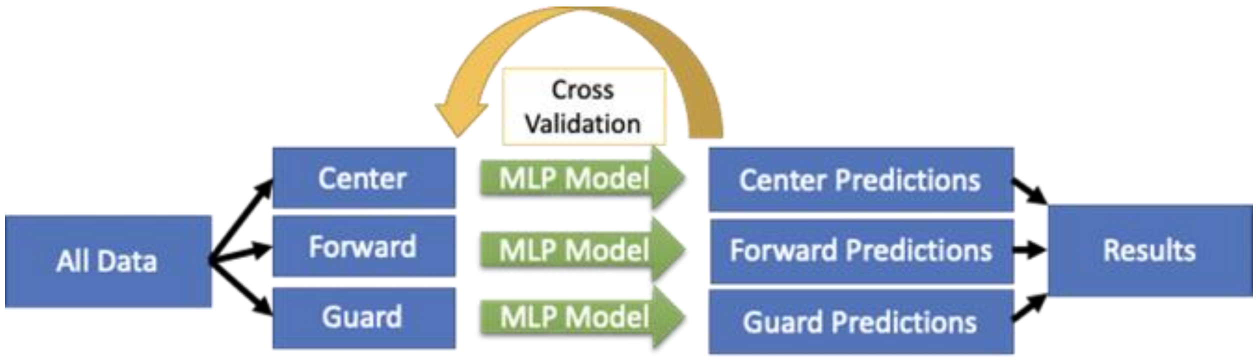

Figure 4: An outline of the overall approach of the segmented model.

The segregated model uses cross validation to assess and develop legitimate predictions for every entry in the dataset. The specific type of cross validation used was n-fold cross validation, with the number of folds chosen to be 5. N-fold cross validation works by dividing the provided data into n equal sized segments. Each fold is used as a testing set and the remaining n-1 folds are used to train the model. This allows for predictions to be generated on every entry in the dataset, while making sure that there is no data leakage between the training and testing datasets.

Because we now had 3 unique datasets, it was thought that feature selection would be beneficial for each dataset independently. This was the case, as the feature selection process isolated different features for each dataset. Feature selection was done using the sklearn SelectFromModel function, with a random forest regressor as the core estimator. Some interesting finding from this was the center dataset had the smallest final feature set, which focused on college performance and mock draft results. The forward dataset was more focused on physical ability (strength, speed) and was less concerned with mock draft rankings. The guard dataset had the most features and only a few biometric columns were removed.

The specific neural network to be used within the segregated model was a 5-node multilayer perceptron with dropout on the input layer, and is shown in Figure 5.

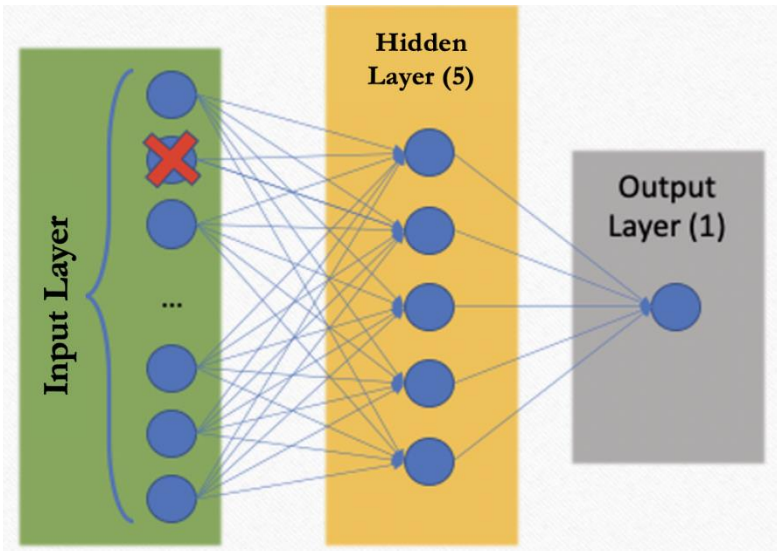

Figure 5: A description of the neural network used within the segregated model. The red cross in the first layer represents node dropout.

The size of the input layer was determined by the specific dataset in question. This is because the feature selection process mentioned above resulted in a unique feature set for each position's data. The size of the hidden layer was arrived at through experimentation, and found to be most effective with around 5 nodes. This was consistent across all datasets. A hidden layer was justified as it was assumed that there would be considerable non-linear relations in how the feature set predicted the regression target. This was found to be the case, as adding a hidden layer often improved performance. Lastly, a single output layer with a linear activation function was chosen to deliver the regression output of the model. This was the obvious choice, as we are looking to predict a single continuous variable and a linear activation function would best translate the actual output and training changes within the model to a direct change in regression output. All other layers had a rectified linear activation function, as this was found to provide the best accuracy. Other options considered included linear activation and a sigmoid activation function. The optimization algorithm of the model was also decided upon through trial and error testing, and was chosen to be the Adam optimization algorithm. Other algorithms considered include Adamax, SGD, and RMSprop. Another important aspect of this model was to select an appropriate loss model. Because we are predicting a continuous variable, the loss function was chosen to be mean squared error (MSE). Other loss functions were also entertained, including mean squared logarithmic error, mean absolute percentage error, and mean absolute error, but MSE provided the best performance.

The last and most influential parameters that were considered when developing these models was the number of training epochs and training batch size. Batch size refers to how many samples should be trained over before the network weights will be updated, and the number of epochs is how many full passes through the training data should be performed. Increasing batch size to higher numbers helps the gradient training changes avoid noisy movements, however batch sizes too big will cause training to be erratic and only move in large jumps. The best batch size for this problem was found to be ~20 samples, or ~3% of the entire dataset. The number of epochs was also determined experimentally. Too many epochs allowed the model to overfit, and too little would not provide the model with enough training sessions. The best number of epochs was found to be around 30.

Lastly, my model investigated the use of oversampling to help improve the final predicted values distribution. Initial results found that the model was unable to predict the higher Wins Made values within the dataset (which went as high as 11), only predicting under 3 for the entire dataset. Oversampling was done using random oversampling, meaning that during training the model was exposed to random duplicate entries from the under-represented class. Usually oversampling is implemented for classification problems, however it was utilized for this regression problem by defining the undersampled classes to be entries with Wins Made values under 0.5 and over 2.5.

SELU Deep Neural Network With Binned Classifications (Alan Dimitriev)

The other three models produced by my group mates were designed as regression machines, in an attempt to increase diversity between our models I decided to approach the problem as a classification one.

From the initial dataset that contained 59 input features, I performed feature selection focusing on removing features that had low variance, high correlation and removing features that had little to no impact on prediction results. As a result, the model I implemented takes in 29 input features. I implemented a training-testing split using a ratio of near 8:2 resulting in 533 input rows for the training set and 130 input rows for the testing set. The data was shuffled during this process since the original dataset is ordered by year and draft position.

The problem being addressed (predicting NBA wins-made or WM) was assessed by using data that was neither sequential, nor spatial, and as a result a deep multilayered perceptron was implemented as the ideal model above either a convolutional neural network or a recurrent neural network.

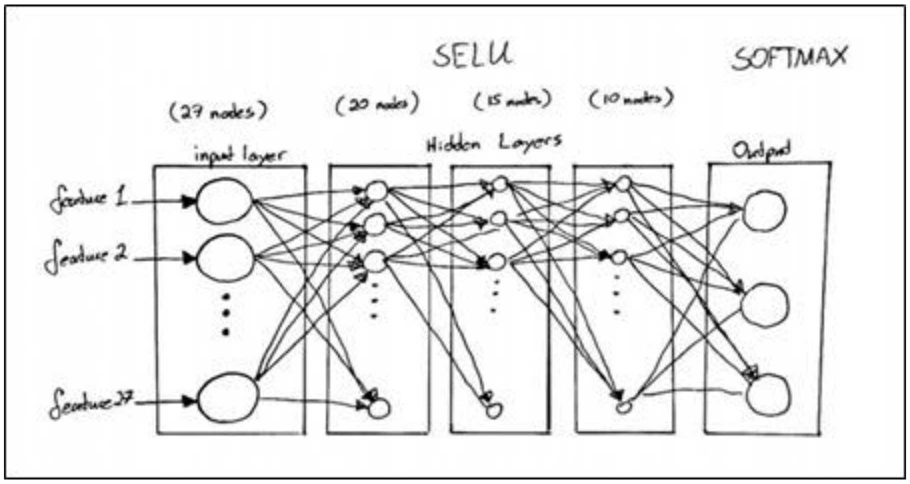

Figure 6: This sketch displays the overall model architecture that was used to implement the predictor.

The model implemented contains five layers: an input layer, three hidden layers, and an output layer, which is displayed in Figure 6. The number of hidden layers was experimented with but ultimately a three-layer system was implemented that had a descending nodal architecture. Since the data was not linearly correlated a non-linear activation function was required. As “relu” was already in use by one of my peers I implemented a “scaled exponential linear unit” (selu) activation function. The benefit of selu is that it is self normalizing, however it does require its own specific weight initialization (achieved by using a Lecun Initializer). The model also implemented a dropout function on two of its hidden layers, both dropping 20% of their nodes randomly during training. This was done to prevent overfitting and increase accuracy variance. While testing different optimizers and trying to differentiate my model from my group mates I settled on the use of either the “Follow the Regularized Leader” or “Nadam” algorithms. During testing I found that the Nadam optimizer outperformed the Ftrl optimizer and decided to use that as my final optimization function (this was decided after receiving feedback from our TA over concerns with the initial use of Ftrl).

When deciding on an output function I did research into what was available through Keras and settled on using a “softmax activation function”. Softmax doesn’t return a binary value such as the classic sigmoid function, but instead returns a class vector that contains confidence probabilities ranging from zero to one for each output class with the entire vector summing to equal one. This is beneficial as it allows the network to always output a prediction and attempts to allow the model to avoid overfitting one specific class.

Due to variance in the model introduced by the dropout function and weight initialization a system was devised where twenty separate models were trained on the same training data, with the model that produced the highest testing accuracy being selected as the final version of the trained model. Each model was run for fifty epochs, this number was selected as it produced the highest accuracy score.

Regarding classification after testing multiple binning scenarios I decided to use three bins to represent whether a player had a negative, positive, or neutral effect on their team's wins (based on their wins made). A player had a negative impact on their team if they produced a WM below zero, a neutral player was a player who had a WM value between 0 and 1.5, and a player had a positive effect on their team if they had a WM value above 1.5. The decision to use three bins was due to a low standard deviation in the dataset, making it very difficult to predict more focused binning scenarios.

Regressive Multilayer Perceptron Neural Network (Adam McCaw)

This model architecture consisted of a 4 layer multilayer perceptron with two hidden layers, an input layer and an output layer. A perceptron architecture means that it is a feedforward neural network, where each node has an input from every node in the previous layer (ignoring dropout). A diagram of the model architecture is shown in Figure 7 where the numbers specify the number of nodes in a layer and the activation functions of each layer are shown.

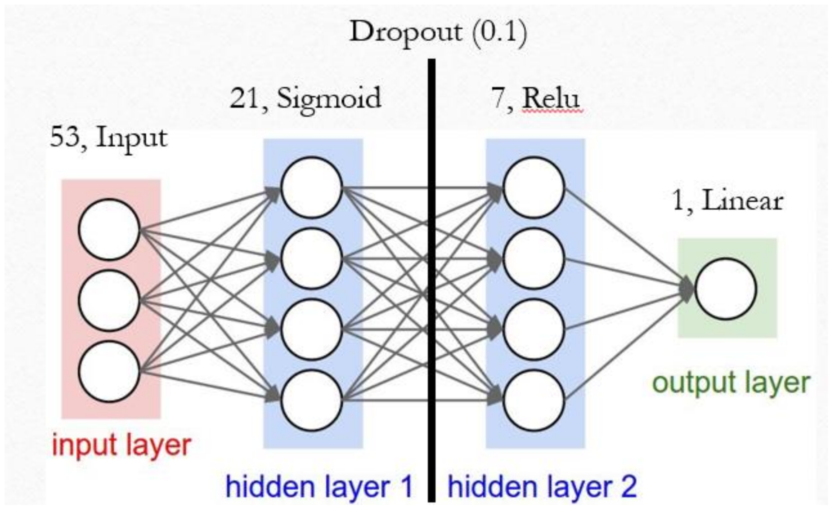

Figure 7: Architecture of regressive 4-layer MLP including node numbers on each layer and activation functions.

The input layer has 53 inputs unlike some of the other models since in this case all input features were included, and feature selection was not done. This was done to allow comparison between other models made by the group that perform feature reduction. Additionally, the model runs quickly and ideally weights of unimportant features will be trained to have little contribution.

Two hidden layers were chosen with a dropout in between them. While architectures of one, two, three, and four hidden nodes were all tested, it was determined that two hidden nodes had the best performance. The first hidden layer consists of 21 nodes (chosen due to best testing results), each with a sigmoid activation function. This layer functions to group features into feature groups and the sigmoid choice allows for each feature group neuron to either be active or inactive. This allowed the neural network to be more generalized compared to using a relu or other activation here, since this would allow more overtraining on the training data. This was found to be true in practice, and sigmoid performed better than relu for this layer. A dropout of 10% was used between hidden layer 1 and 2, meaning a random 10% of hidden layer 1's nodes were chosen not to connect to the next layer. Implementing this dropout improved the results and reduced overfitting, but increasing to above a 10% dropout lost too much of the information and hurt the model. The second hidden layer consisted of 7 nodes (chosen due to best testing results), each with a relu activation function. A relu activation function was chosen here to allow better variability and scalability in the final output, since if sigmoid was chosen here, it would provide a more discrete output with near-binary output from each hidden node, where a continuous scalable output is required for WM. This theory was proven true since relu had better performance than sigmoid in testing. It also performed better than linear due to its ability to create complex curves in the feature plane compared to the linear separation done by linear activation. While the 21, 7 node numbers were chosen based on testing it makes sense that these selections performed best. If too many hidden nodes are used it would allow overfitting on the training data. However, if too few were chosen, this model would not have the variability and scalability required to implement the regressive model.

The output layer was one node with a linear activation function that was used to predict WM regressively. The linear activation function was chosen to allow the output to scale linearly through the desired WM range of ~ -1 to 12.

Backpropagation with the Adam optimizer was used for weight adjustments in the model since it is the standard for MLPs and performed the best in testing. The weights were initialized to be random between zero and one, and biases were included at every layer. The loss function being reduced was chosen to be mean squared error (MSE) since this is also the evaluation metric chosen, so we want this to be minimized. MSE was chosen for these since it is a good measure of how close output values are to their expected values.

The input data was normalized using standard scaling normalization, which is a standard choice. Normalizing the data also resulted in improvement in results since the weights were not required to train as high or low depending on the scale of the inputs in the feature, allowing for improved training. The data was then split into 15% testing, 15% validation and 70% training, which is a standard for neural network training. The model was implemented using keras, with scikit learn used to split the data.

Batch size was chosen to be 0, meaning each weight was changed for every training sample, which performed better than batch training and was still quick to run. The training was completed over 100 epochs. This was chosen because increasing epochs worsened results (higher MSE) due to overfitting on the training data. Less epochs did not provide enough training and also resulted in worse results. Additionally, since the number of epochs and hidden nodes is relatively low, the neural network provides the advantage or running quickly.

Results

Radial Basis Function Network (Hayden Wang)

The model on average over multiple tests with the same parameters achieved a MSE of 0.015 on the normalized data which the square root of the MSE of the inverse_transform() rescaled that value to 1.67 to be able to easily compare with the other group members’ results. I was only able to train the model for 100 epochs because I found that any more and it would overfit while less resulted in drastically higher MSE values. For reference, the MSE on training data was 1.63 which is decently close to the MSE of the testing data (1.67).

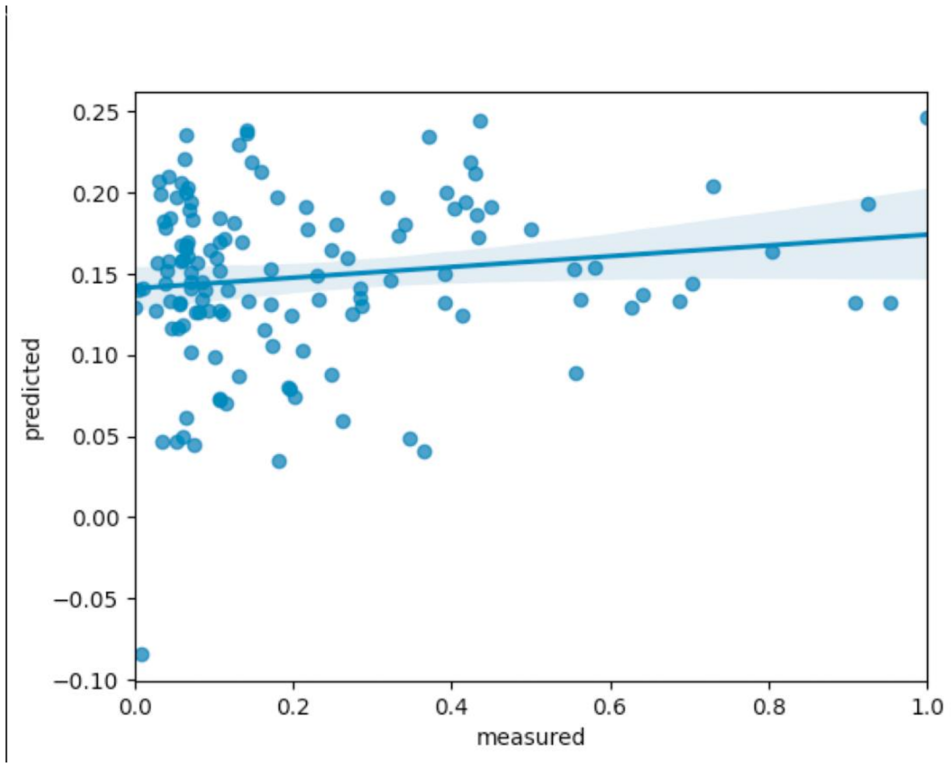

Figure 8: Graph comparing the wins-made predicted by the model to the actual values.

From the figure you can see that the model frequently predicts a wins-made of around 0.2 (target was normalized to be between 0 and 1). Like many of my group members mentioned, this is most likely due to the lack of sample data that had high wins-made. This is sometimes the reality of the type of data we are looking at since there a proportionately much fewer all-star level players than there are average ones. The model looked at those high-performing players as outliers and decided that predicting for the majority would result in a lower total error.

Segregated Multilayer Neural Network with Dropout (Noah Rowe)

The final segregated model, without oversampling, was able to achieve a mean squared error of 2.25 Wins Made. With overfitting, the mean squared error increased to 2.60 Wins Made. Figure 8 shows an average of the training error across all cross validated models for each of the datasets, as well as the final mean squared error of each model.

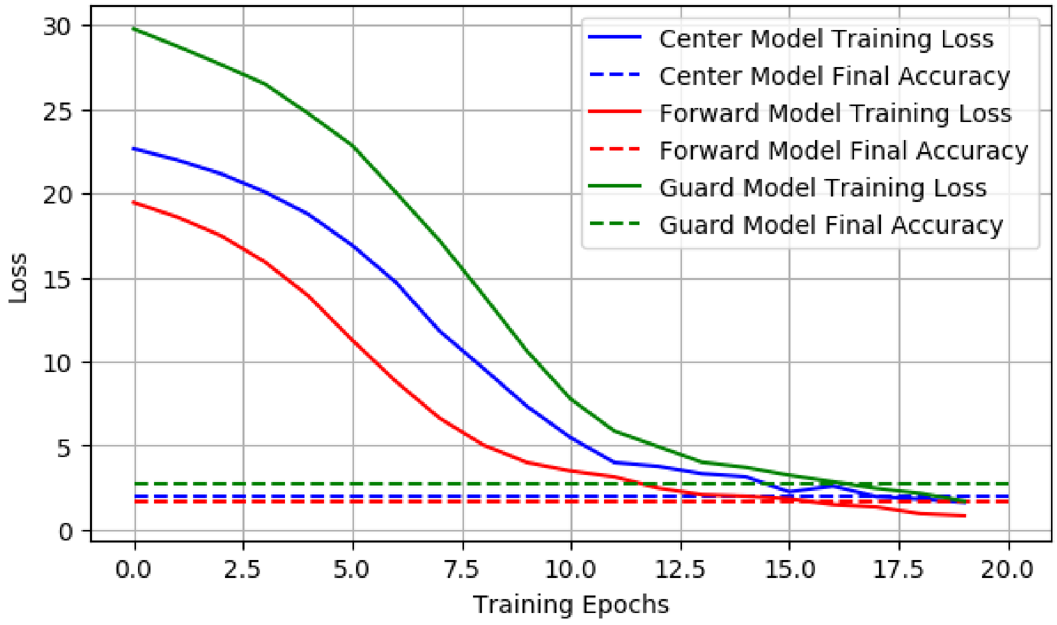

Figure 9: A plot showing the training error trends as training epochs increased. The dotted lines represent the averages of the testing accuracies for each model across all cross-validation folds.

Due to the structure of n-fold cross validation, where each model is only evaluated with one testing set at the end of model training, there was no opportunity to plot validation accuracy over training epochs. This would have been very useful to investigate, as it would have provided useful insight as to when overfitting began to occur. In the future I will try to design my models to allow for this.

All final accuracy values are outlined in Table 2. Combined results are calculated by evaluating the mean squared error over the combined individual predictions, as this accounts for class size.

Table 2: A summary of model accuracies under different conditions.

Model Type		**Results (mean square error)**
	Center Model	2.17

Without Oversampling	Guard Model	3.09
	Forward Model	2.00
	Combined Results	2.23
With Oversampling	Center Model	2.53
	Guard Model	3.22
	Forward Model	2.17
	Combined Results	2.66

As shown in Table 2, oversampling generally hurts model results. However, it was noted that oversampling strengthened the positive relation between predicted and true values across the dataset. This is shown by the linear trend line in Figure 9 and Figure 10. Figure 9 shows the relationship between predicted and actual values for non-oversampled training and Figure 10 shows the same for oversampled model output.

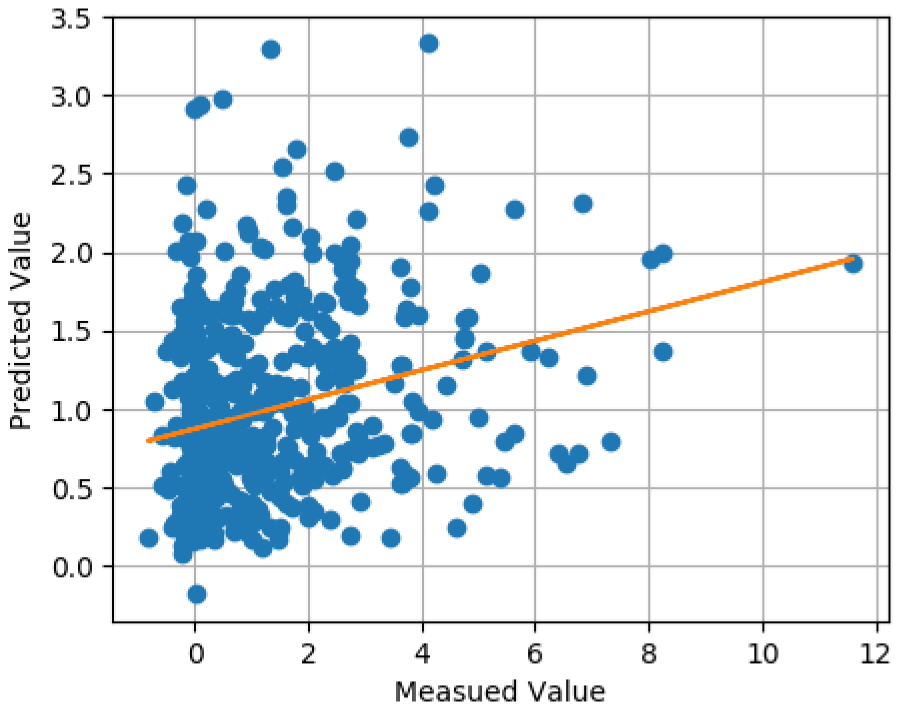

Figure 10: A plot that shows the relationship between measured and predicted Wins Made values, as well as a fitted linear trend line (m=0.0936, b=0.872)

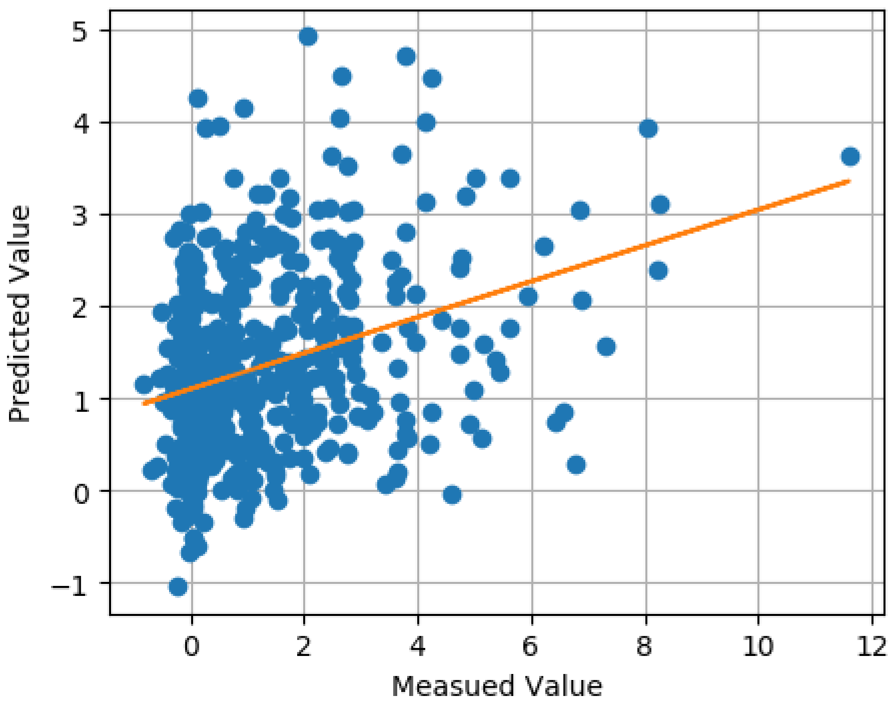

Figure 11: A plot that shows the relationship between measured and predicted Wins Made values, as well as a fitted linear trend line (m=0.184, b=1.10) for the oversampled models.

An interesting aspect of these plots is that the standard deviation of the true wins made values is 1.62. The standard deviation of the predicted values is 0.57 and 0.97 for normal training and oversampling training, respectively. This is evidence that oversampling is helping to replicate the actual distribution of the target variable, despite a lower mean squared error. Also, the slopes of the linear trend lines fit to Figures 9 and 10 show that the oversampling model creates a stronger relationship between measured and predicted values than the normal model does. This is seen in the slope of each line, where the oversampling data has a slope closer to 1 than the normal models. It also makes sense that the oversampled data has a larger offset (b value), as it has been trained on higher target values in general.

SELU Deep Neural Network With Binned Classifications (Alan Dimitriev)

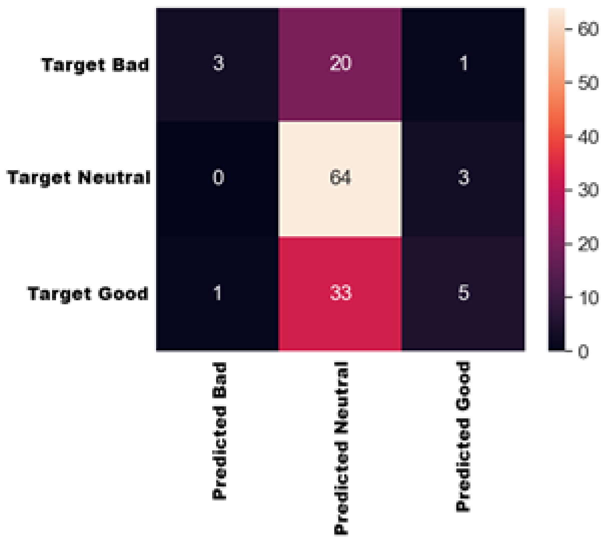

Figure 12: This figure displays a confusion matrix generated based on the predictions outputted by the model compared to the target values, with "Good" representing a positive WM and "Bad" representing a negative WM.

The model did not manage to produce a seemingly acceptable accuracy as the highest accuracy it was able to generate was 54.1%, the confusion matrix can be seen in the Figure 11. The model tended to predict most players as having a neutral WM value. The precision and recall for each class are displayed in the table below.

Table 3: This table displays the resultant precision and recall for each of the three possible output classes.

Class	**Precision**	**Recall**
Negative (Bad)	75%	12.5%
Neutral	54.7%	95.5%
Positive (Good)	55.6%	12.8%

The model implemented can correctly classify neutral players at a rate of 95.5% but is unable to correctly classify players when they are either bad or good. This can be attributed to the structure of the dataset itself. Only 9% of players logged in our data have a WM over 3, this results in the model not being able to properly train to detect these values. As a result, the model incorrectly predicts many players that have WM's that would place them in either the positive or negative bin as neutral. When looking at the models produced by my groupmates, we see that their regression models generate an MSE of between 1 and 2. When trying to classify an already limited dataset in regards to the extreme values on a scale between -1 and 12 it is easy to see where the model would have difficulty generating predictions if the MSE is enough to influence predictions by a unit of 2, which can result in players with negative value being predicted as neutral or even positive and the same can be said for players with positive WM when our error could result in them being classified as having a neutral or even negative impact on team wins.

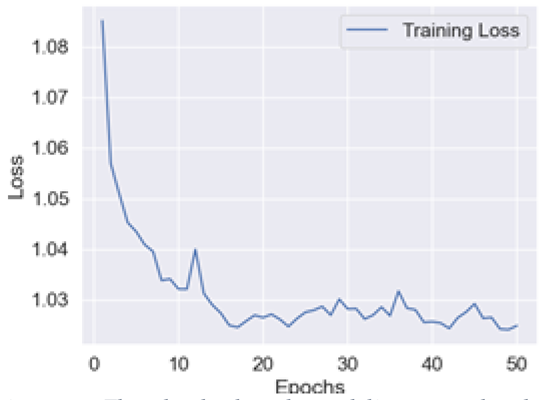

Figure 13: This plot displays the model's training loss by epoch.

As can be seen in Figure 12, the model itself does decrease its loss value as it iterates through the epochs, but eventually it stabilizes and does not produce any notable improvement. This can be attributed to the problem before: the dataset simply does not contain enough valid examples of players with high WM values. However, it must be considered that in real basketball the majority of players do not have impressive statistics which is why we have "star" players that teams are built around. The potential exists to remove these high-level players from the dataset to allow for the model to try and classify the remaining players more accurately (with new binning definitions) however this is unrealistic and doesn't fit our target goal as star players do exist and the real application of models such as ours is to use the model to find players with star potential, or at the very least players that will have a positive influence on the teams wins. From a design standpoint however, the architecture of the model implemented is suitable for the task. The model is limited by the data it has available to it and would perform better with a larger dataset that included more players with positive or negative WM's, as well as a larger dataset overall as 663 input rows is a very small amount of data when dealing with deep neural networks.

Regressive Multilayer Perceptron Neural Network (Adam McCaw)

This regressive model that predicted a numerical "wins made" value was evaluated using mean squared error (MSE), where the lower the MSE, the better the model. Many different models were run on the training and testing dataset, but the final model described in the implementation section had an MSE lower than the other tested options of 1.68 on the testing dataset. On the training data the model had a 2.24 MSE, indicating the model was not overfitted. A residuals plot showing the predicted minus the desired value is shown in Figure 13.

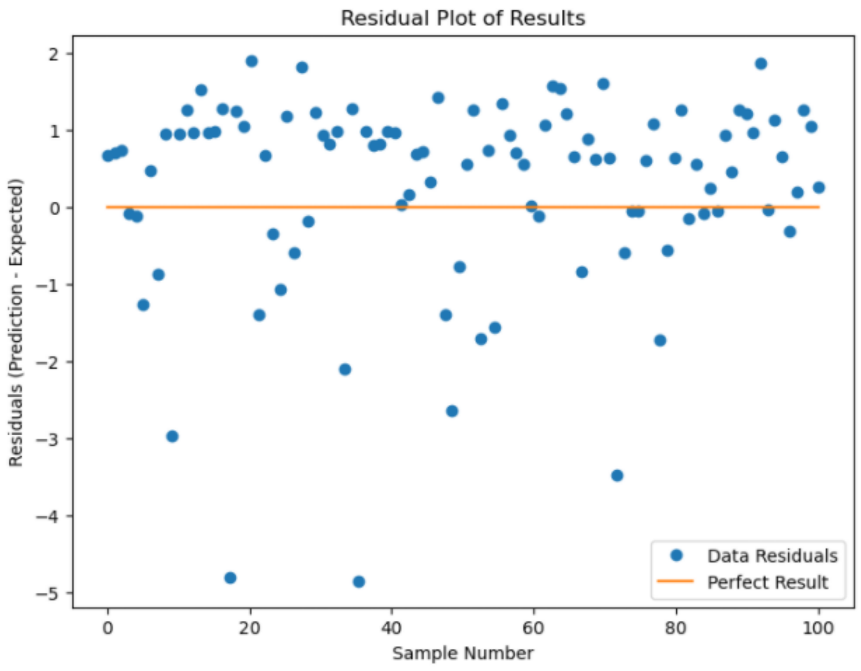

Figure 14: Residuals plot of Results with predicted – expected on y-axis and sample number on x-axis.

Additionally, a plot was made showing expected vs predicted values shown in Figure 14.

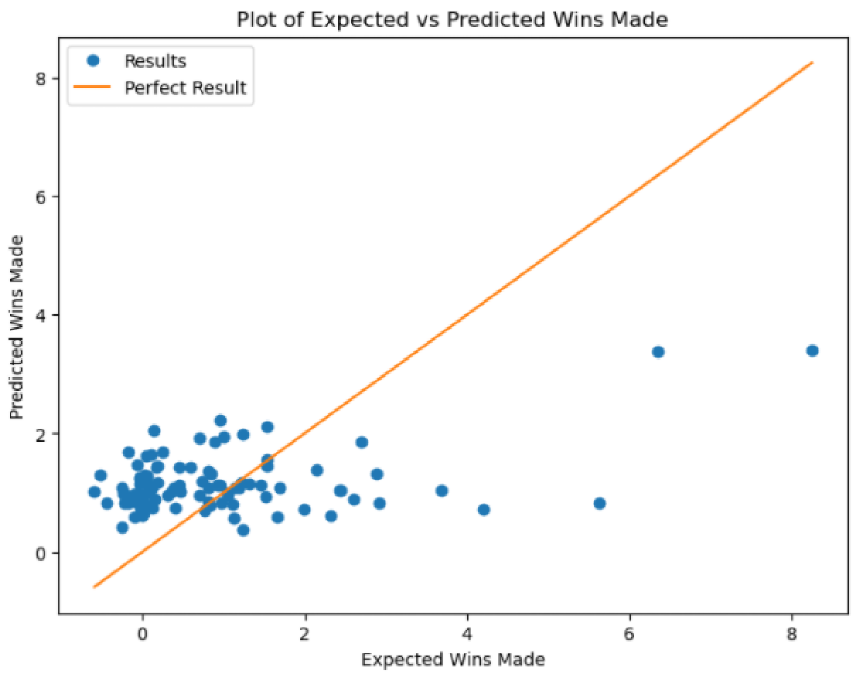

Figure 15: Plot of Expected WM vs Predicted WM for 4-layer MLP.

The mean squared error over training time was also plotted. This is shown in Figure 15.

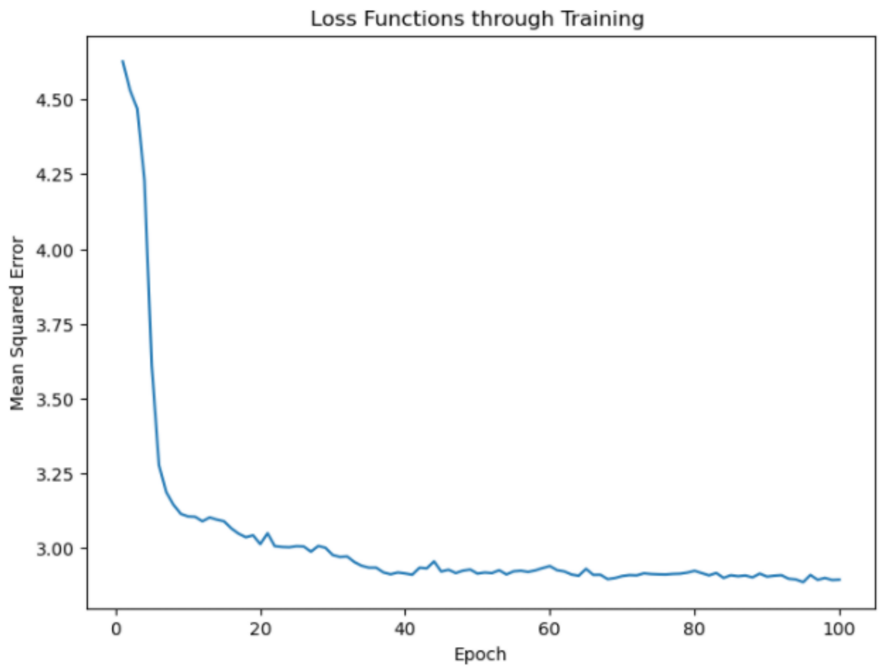

Figure 16:Plot of model loss over epoch iterations using MSE as loss.

Figure 15 clearly shows this model is successfully training the dataset, as the mean squared error decreases dramatically from the first epoch to the last.

These results came from the model being run on one grouping of shuffled testing and training data. However, the data can be shuffled and split in many ways, so to determine how this model performs across many different dataset splits, once the final model was determined it was run 50 times on newly shuffled training and testing sets each time. This provided an average MSE of 2.26, indicating the dataset split that I originally ran the data on was an above average version. However, the minimum MSE for these 50 was 1.13, indicating my original split was not the best split for this architecture. This also had an average training MSE of 2.10, so the model was again not overfitted to training data.

Overall, the results of an MSE of 1.68 and an average MSE over 50 evaluations of 2.26 indicates that while the model did train, it did not reach a point where it predicted the WMs well. Since an MSE of 1.68 is quite large when the range of WMs is approximately -1 to 12, and most of them are in the 0 to 1 range. However, the results from this model were like the results of models from other group members. Figure 13 and Figure 14 both show that samples that were the hardest to predict were the ones that had higher WMs. This is likely because the dataset has very few of these high values and has many values at low values between 0 and 1. Thus, these high WM values would not have changed the weights enough during training to predict them well, since the training is dominated by low WM values. The model also predicts higher values than desired for many of the very small WMs close to 0, as shown in Figure 13. This is likely because the scale of WMs goes from very small values such as 0.01, 12, so training a model to predict both can be difficult. Additionally, the high MSE is likely caused by the low number of training samples. Multilayer perceptrons with many hidden nodes often require thousands of training samples to be trained properly for the data. However, since our group built the dataset ourselves, it was small (75% of 633 samples) which did not provide enough data to train the model well. This small dataset also meant the results were very dependent on how the data was split, which is why the MSE massively fluctuates for my model with an MSE standard deviation over the 50 evaluations of 0.58.

If this project were to be repeated, I would add feature selection to my model which removes unnecessary input features based on high correlation or low variance, which would ideally improve the model. Additionally, while an exponential activation was tested for the output node and did not improve the model, an exponential model should be considered in more depth, since the scale varies from very small to quite large values and an architecture that scales exponentially could theoretically model this better.

Conclusion and Open Problems

Overall, our project used 4 different implementations for predicting wins contributed "wins made" by an NBA player to their team in their first season of NBA play based on their college basketball and their NBA combine stats. One model implemented separated the data by position into forwards, centers and guards, and ran three separate regressive 3 layer multilayer perceptrons (MLPs) to predict WMs on each of these separately. This model had an MSE of 2.26. This model had a slightly higher MSE than other models implemented, likely due to splitting the data into 3 smaller sets. The second model of a 5-layer MLP that implements classification, got an accuracy of 54.1%. The third model implementation used a regressive 4-layer MLP to predict WM. It differed from the first model in that it predicted all of the positions together in one MLP, and it had an MSE of 1.68 (or 2.26 when averaged over 50 evaluations on different training and testing data splits). The fourth model implementation was a radial basis function network. This model had 3 layers, where it's hidden layer had as many nodes as there were training samples. This was also a regressive model that predicted WM, and its MSE was 1.67, performing similarly to other models.

All four of these models showed success training on the dataset, lowering the loss function through iteration. However, none of the models had useful results for predicting WM with a 54% classification accuracy and a minimum MSE of 1.67 for WMs in a range of ~ -1 to 12. All these models also struggled to predict extreme WM values due to the scale of the dataset. However, all the models implemented are multilayered neural networks that can require a lot of data to train well. With a dataset of only 633 samples, the training is unlikely to be exceptional and the chosen training and testing set will have a significant effect on results. Additionally, when stats were missing, they were filled in with the average in the category, which often does not reflect the actual stats of that player and can be very wrong. This assumption limited the accuracy of the models, which trained incorrectly for some of these samples. To improve on these models in the future the first step would be to build a larger dataset, without any averaging of stats. This would provide a larger, more accurate, dataset that would allow better training of neural network architecture. This would also help improve determination of useful versus unimportant features, which would allow the neural networks to be optimized. Ideally, with these steps, these models could be used to predict "wins made" accurately for college basketball players, helping NBA teams determine which players should be drafted.