Table of Contents:

---

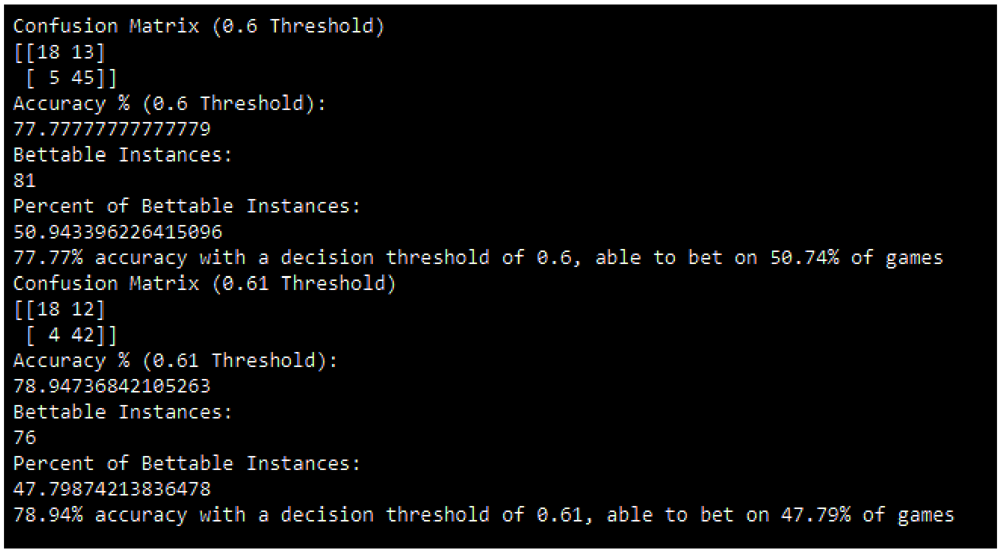
1. High-Level Summary

Goals:

1. Predict the whether the favorite will win an NFL Football Game
2. Interpret & Explain

Part1:

- Deployed a neural network via Keras API
- After tuning the model, we were able to correctly predict 77.77% of game outcomes with a 0.60 classification threshold & 78.94% with a 0.61 classification threshold.
    - Using classification thresholds, we were able to place bets on 47%-50% of games.



```
Confusion Matrix (0.6 Threshold)
[[18 13]
 [ 5 45]]
Accuracy % (0.6 Threshold):
77.77777777777779
Bettable Instances:
81
Percent of Bettable Instances:
50.943396226415096
77.77% accuracy with a decision threshold of 0.6, able to bet on 50.74% of games
Confusion Matrix (0.61 Threshold)
[[18 12]
 [ 4 42]]
Accuracy % (0.61 Threshold):
78.94736842105263
Bettable Instances:
76
Percent of Bettable Instances:
47.79874213836478
78.94% accuracy with a decision threshold of 0.61, able to bet on 47.79% of games
```

Overall Summary:

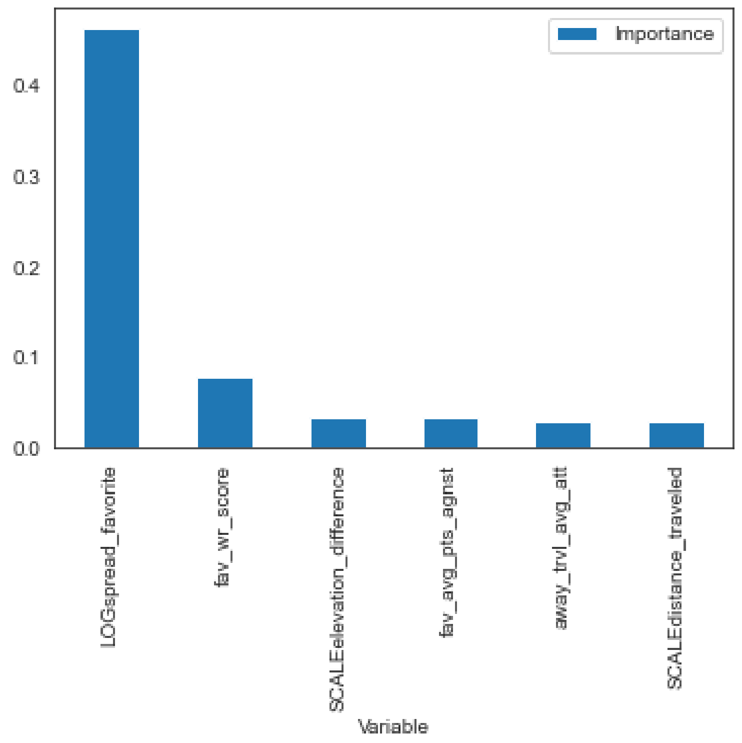
- The model performs well. There are no issues with respect to only predicting one class, so this model may have more value regard general usage (when compared to the XGB discussed)
- It still has a higher accuracy when predicting "fav_won" as the outcome, but the model does not "fall-off" when the class predicted changes.
- The most valuable features in the model were the Las Vegas Spread (Log) & the Favorite WR score.
    - Because these two features aligned with the finding from the XGB results (SHAP plots & PDPs), we can be more certain that these two features carry a large amount of weight in indicating the predicted outcome,

Part2:

Permutation Feature Importance

- According to permutation importance, the most important factor in determining the outcome was LogSpread, which is the log transformation of Las Vegas spread.
    - This was done to capture the relative versus absolute differences in the values. (2-point spread from 2-4 versus a 2-point spread from 14-16)

- The next most important feature was fav_wr_score, which was the difference between the favorite teams WR rating & the underdogs SCDY rating.



---

2. In-Depth Explanation

Preprocessing
- Packages are loaded
- Data is loaded
- Shuffled the data
- Data is further cleaned & split

Variable Selection
- Used Random Forest
- Based on Gini/Impurity Importance and Permutation Importance
- Combined the features selected by each variable

Gini/Impurity Importance:
- Determines feature importance based on Mean Decrease of Impurity (Decrease of Impurity = Purity Gain)
- Measures the decrease in Gini Impurity when said variable is chosen to split a node

- Impurity is a measure of how often a randomly chosen observation would be incorrectly labeled according to the distributions because of the split
- If the Impurity decreases as a result of the split, then the variable is more important. Thus, a larger decrease in impurity indicates higher variable importance.
- However, impurity-based importance tends to overestimate the importance of numerical features (scikit-learn.org)
- To address this issue, we also use permutation importance as an additional measure.

Permutation Importance
- After the model is trained, a column is randomly shuffled
- The decrease in model accuracy is compared before & after the random shuffle
- Permutation importance ranks variable importance based on this measure

Preprocessing
- Scaled the data

Sequential Neural Network Model:
- Sequential Model: A plain sequential stack of layers for the neural network. Each input/output has exactly one layer
- Functional Model: A model that does not need to attach layers in a sequential order.
- Built a sequential model because of familiarity & reference ability

---

General Model Summary
- In the model, there are layers.
  - Input Layer - Where the input data is "fed" to the model
  - Hidden Layer - The "black box" part of the model
  - Output Layer - The model returns
  - Dropout Layer - A layer which cancels out some of the neurons
  - Dense Layer - Hidden layers. They receive input from all neurons in the previous layer.
    - Used for changing the dimensionality

- In each layer, there are nodes/neurons that are essentially just a set of inputs/weights & an activation function
- The neurons can also be thought of as a "bucket" for the inputs (which are the outputs of the previous layer)
- The neuron then passes a signal, which serves as the input for the next layer.
- Models differentiate, and different weights/activation functions/number of neurons/etc can influence the performance
- So, we can tune these different parameters to achieve the best model
- No dropout layers were included

Sequential Model:
- L1 Units: 256
- L1 Bias: True
- L1 Activation: Swish
- L1 Initializer: He Normal
- L2 Units: 256
- L2 Bias: True
- L2 Activation: Swish
- L2 Initializer: Glorot Uniform

```
        - L3 Units: 128
        - L3 Bias: False
        - L3 Activation: Swish
        - L3 Initializer: He Normal
        - Learning Rate: 0.001


Best val_loss So Far: 0.6102414131164551
Total elapsed time: 01h 35m 06s
INFO:tensorflow:Oracle triggered exit
```

Model Summary/Parameters

Layer 1:
Units: 256
- The dimension of the output space (neurons)
- Essentially the output size of a layer

Bias: True
- Adds a constant to the activation function
- Can be thought of as the constant in a linear function
- Used to add flexibility with respect to the activation function

Activation Function: Swish
- Neurons interacting via a linear function is often too simple
- Activation functions introduce nonlinearity to the network
- Determines which neurons should be activated
- Swish Activation Function
    - Helps address the "dying neurons “issue which occurs with Relu activation functions
    - This occurs with Relu because its derivative is 0 for half of the X input values
    - If the gradient for a parameter is 0, then the parameter will not be updated
  - Swish is differentiable at all points, so a derivative will always exist (unlike Relu)
  - Swish Function: $f(x)=x\sigma(x)$ where $\sigma(x)$ is a sigmoid function
  - Swish can be viewed as a function where its non-linearity fluctuates between a linear function and the Relu function depending on the beta value
    - Beta can be constant or trainable parameter
  - Regarding the gradient of a Swish function, it is unbound.

Kernel Initializer: He Normal
- Determines which distribution/function is used for initializing the weights
- The network must start with weights & then update them to more appropriate values
- The initializer sets a basis or starting point for the weights based on some distribution
  - He Normal
    - Usually used with Relu activation (Makes sense since Swish is similar to relu)
    - Bases/sets the initial values of the neuron weights to a normal distribution
    - Difference between Glorot Normal & He Normal is that the variance of the weights is multiplied by 2 in He.
    - This is because a larger variance spreads out the distribution of weight values, which leads to more robust spread of values
    - He also tends to provide similar variance through all layers

Layer 2:
Units: 256
- See above

Bias: True
- See above

Activation Function: Swish

- See above

Kernel Initializer: Glorot Uniform

- The initial neuron weights are based on a uniform distribution (they are all the same)
- Draws samples from a uniform distribution, within the [limits]
- Within the limits [-limit, limit] (tensorflow.org)
    - Limit being sqrt(6 / (fan_in + fan_out))
- In our case, limit = 0.108253
    - limit = sqrt(6 /256+256) = limit = sqrt(6 /input units + output units)

Layer 3:

Units: 128

- See above

Bias: False

- See above

Activation Function: Swish:

- See above

Kernel Initializer: He Normal

- See Above

Learning Rate: 0.001

- β in the AdaGrad description below
- It is the constant learning rate divided by Alpha+Epsilon (See AdaGrad section below)

Optimization:

Model Optimizer: AdaGrad

- Adaptive Gradient Optimizer

Gradient Descent:

- Essentially just an optimization function to minimize loss(or cost)
- Is based on a convex function, where the minimum is the global loss minimum
- We start at an arbitrary point to evaluate performance
- We find the derivative(slope) - from there we can use a tangent line to observe the steepness of the slope
- The steepness of the slope indicates where we are in the convex function & informs us how to update the parameters (weights)
- The steepness should gradually reduce until it reaches the lowest point on the curve(point of convergence i.e. the minimum)

Problems with Gradient Descent:

- Sometimes finds local minimums or saddle points
- Vanishing Gradients
    - The gradient becomes smaller & smaller in each step due to repeated back propagation on many layers in the network
    - Backpropagation calculates the gradients(slope/derivative) of a loss function by the chain rule (below)
    - Different activation functions attempt to address this issue

Chain Rule: $f'(x) = g(h(x))' = g'(h(x))*h'(x)$

Main function is equal to plugging h(x) into function g(x)

To find the derivative of f(x), we can take the derivative of function g & multiply it by the derivative of function h.

- The gradient(slope, derivative) can become smaller & smaller due to the derivative calculation in the back propagation process (especially when there are many layers)

    - It can become smaller & smaller because we repeatedly multiply the gradient by a number <1 - happens for each layer
    - Thus, the step size will approach 0 & we will never reach an optimum.

This really shouldn't be an issue with a NN of just 3 layers.
But we went ahead and used AdaGrad just in case

Now on the AdaGrad:

- Utilizes different learning rates for different weights & iterations
- This is because there are sparse & dense features (some are mostly zeros-sparse(playoff) & some are mostly non-zeros-dense(logspread))
- AdaGrad weight-update formula with respect to the loss function: $W_t = W_{t-1} - \lambda_t*(\partial loss/(\partial W_{t-1}))$
    - Where W_t = Current weight to be updated, W_{t-1} = previous weight, λ_t = Learning rate, $\partial loss$ = derivative of loss
    - Weight (current) = Weight(current -1) - Learning Rate(current)*(Derivative of loss / Derivative of Weight(current-1))
        - Where λ_t = Learning rate = $\lambda_t = \beta / \alpha_t + \varepsilon$
        - $\lambda_t = \beta / \alpha_t + \varepsilon$ = Learning Rate = Constant (Learning Rate) / Alpha(current)+epsilon
            - Where epsilon = ε = a small constant positive
            - Where alpha current = $\alpha_t = \sum(\partial loss / \partial W_{t-1})^2$
        - So as alpha increases, the learning rate decreases as the iterations go on
        - So the learning rate is changing (decreasing)
        - As the learning rate decreases the weights are adjusted less & less
        - This helps ensure that the weights are efficiently approaching the minimum loss of the convex function

    - This is how AdaGrad overcomes the vanishing/exploding gradients dilemma found in standard Gradient Descent Optimization
    - This is also why AdaGrad was selected as the optimizer in the mode

---

Tuning Optimizer:

Tuning: Bayesian Optimization (keras.tuner.BayesianOptimization)
- Keras' BayesOpt oracle uses a UCB acquisition function. (Whereas we used POI for the xgboost in R)

UCB Acquisition Function:
- Used to evaluate the probability of gain from more exploration
- Contains both exploration & exploitation parameters
- Goal is to balance exploration & exploitation
- $(x;\lambda)=\mu(x)+\lambda\sigma(x)$ (UCB)
- $(x;\lambda)=\mu(x)-\lambda\sigma(x)$ (LCB)
- $\mu(x)$ is the expected performance (exploitation) (mean)
- $\sigma(x)$ is the uncertainty (exploration) (sqrt variance)
- λ is the trade-off parameter (Tunable parameter beta)
- When λ is small, the function will favor solutions that are expected to be high-performing, i.e., $\mu(x)>\lambda\sigma(x)$ i.e. more exploitation, less exploration
- The opposite is true when λ is large. (More exploration $\mu(x)<\lambda\sigma(x)$)

Other Parameters:
- Batch Size: 102
    - We have 1428 training observations, so this will allow for 14 iterations per epoch

Early Stopping: 20
    - If the validation loss does not improve for 20 epochs, then continue to the next trail

<u>Class Weights</u>:

- Weighs the loss function during training
- Helps the learning process by focusing on instances with larger learning errors (with respect to the class weights given) to improve performance

<u>Max Trials</u>: 200

- The number of tuning parameter combinations to be tested