Final Project: Deliverable 2

1. Your objective should be stated in written form. What are you trying to accomplish? Predicting a number? Classifying? Your objective must reference the context of the problem, specifically. (You may reuse the objective from Deliverable 1 or revise as appropriate.)

One of the most special moments for all couples is the moment when they begin to take their next steps towards marriage. Though for this to typically happen, an engagement takes place first. However, even before that traditionally one must go through the exciting process of choosing the special diamond that will be forever remembered in the heart of their significant other. Masters Diamonds is opening this summer and it is our mission to provide our customers a friendly educational environment while offering the rarest and best diamonds at the fairest prices on the market. Since we plan to open in a few months we wish to first discover the right price that we should be setting our diamonds to have a competitive edge over our competitors.

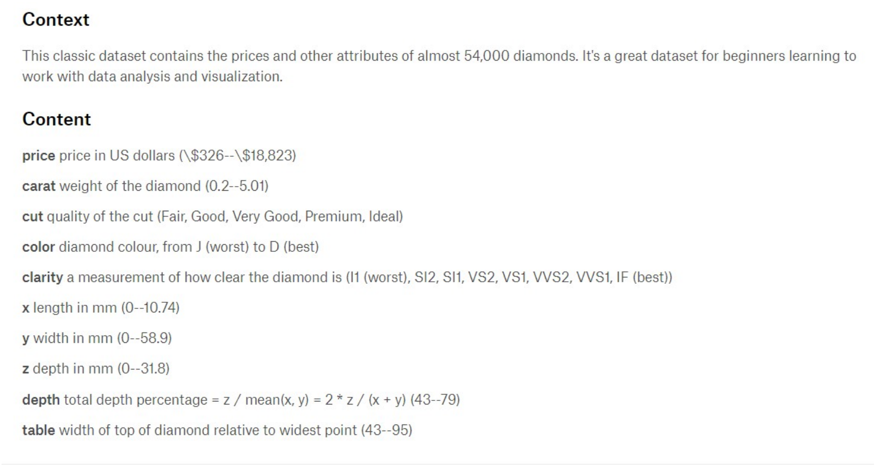
Being a new company, we have reached out to William and Mary's MSBA program for help. All diamonds are created with certain qualities that make them special and we have provided our large dataset containing prices and other attributes of around 54,000 diamonds. For this artificial intelligence project, we are seeking help in planning and creating an ANN model that is aiming to the best of its capabilities to predict the accuracy of the price of diamonds. The model being developed will be a regressor neural network that will help with predicting this price accuracy based on the number of qualities of the data set that is provided.

Overall, we are hoping that this regressor model will allow Masters Diamonds to efficently predict diamond prices based on these attributes, so that we can compare the results with our own diamonds and better market our inventory to our customers when we open for business this summer.

2. The source URL(s) for the data and description should be included.

Within our dataset contains quantitative info on diamond's price, carat or the weight of the diamond, the length, width, and depth of each diamond. Additional quantitative attributes included are the total depth percentage and the table, which is the width of the top of each diamond relative to their widest point. Included in the data are three qualitative variables as well. These include cut or the quality of the cut ranging from Fair to Ideal, color of the diamond ranging from J being the worst to D being the best, and clarity measurements or how clear the diamond is ((I1 (worst), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (best)). Since, these quantitative variables are important towards determining the price of diamonds we recommend on converting them to dummy variables in order to still use them in our required regressor model to see how they help with predicting the price of our diamonds.

Source URL for my project: *https://www.kaggle.com/shivam2503/diamonds*



Context

This classic dataset contains the prices and other attributes of almost 54,000 diamonds. It's a great dataset for beginners learning to work with data analysis and visualization.

Content

price price in US dollars (\$326--\$18,823)

carat weight of the diamond (0.2--5.01)

cut quality of the cut (Fair, Good, Very Good, Premium, Ideal)

color diamond colour, from J (worst) to D (best)

clarity a measurement of how clear the diamond is (I1 (worst), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (best))

x length in mm (0--10.74)

y width in mm (0--58.9)

z depth in mm (0--31.8)

depth total depth percentage = z / mean(x, y) = 2 * z / (x + y) (43--79)

table width of top of diamond relative to widest point (43--95)

3. Your final ANN model, in code, in an attachment.

   (My final ANN model code is listed below after the final question.)

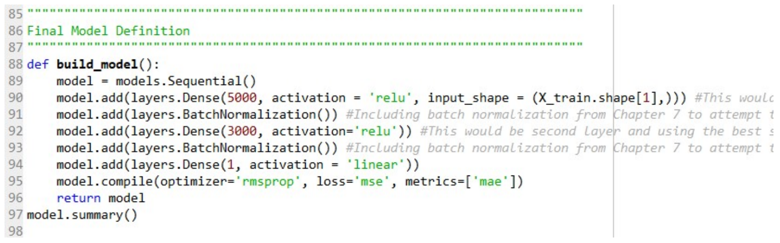
4. Your final model and training algorithm, in words.

   For my final model and training algorithm I wanted to work on creating a regression model to help with predicting future diamond prices based on the input variables that my data set consists of. Before I created my final model I first took my dataset and split the data into X_train, or all the input variables, Y_train or the price of the diamonds, X_test and Y_test using an 80/20 split, which would be used later to evaluate the final model.

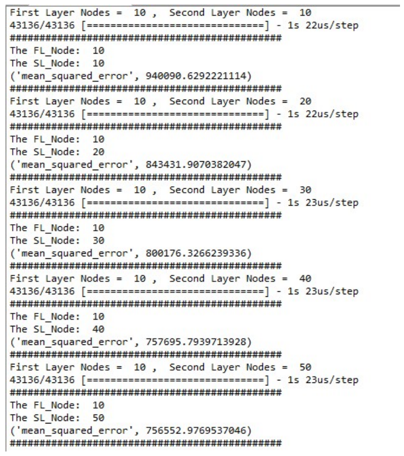
   Since my project is going to be using the model multiple times I used sequential to first build it and then added my layers and dense layers afterwards. For the layer units this took a good while to try and experiment with to find a right value to help lower the MSE and find a good MAE to focus on. I will discuss this more during my experiment plan, but I went with a larger unit value of 5000 for my first layer and then 3000 for my second layer. Originally, I was using much lower units when setting the layers for my model, but I found the MSE to be extremely large as you can see in the images below when experimenting. I also experimented with adding additional hidden and dense layers, but besides the fact that it would take longer for my model to train for the parameters I didn't really get too much of a change in the resulting MSEs while experimenting.

   For the first layer I set the input shape to my X_train.shape index 1 value. For my model compiler I focused on setting the model with the MSE loss function or mean squared error, this was to help me focus on the square difference between my predictions and the final target. Then my metric for my final model is MAE or the mean absolute error, which is the absolute value difference between my dataset predictions and my final target.

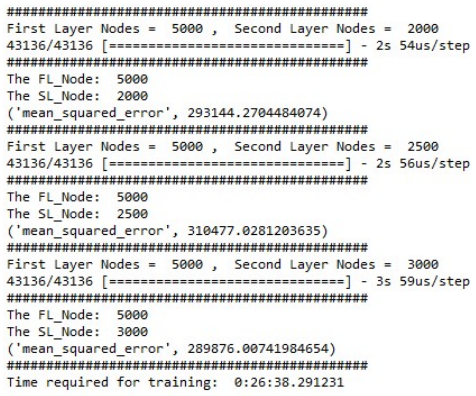
   Once I finished my validation process I then trained a fresh model that was based on fitting the X_train and the Y_train against the X_test and Y_test and saving that to test_mse_score and test_mae_score to view.



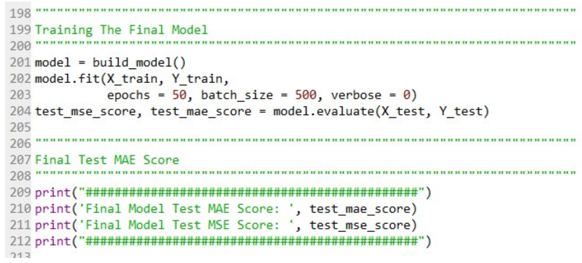
```
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Final Model Definition
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
def build_model():
    model = models.Sequential()
    model.add(layers.Dense(5000, activation = 'relu', input_shape = (X_train.shape[1],))) #This woulc
    model.add(layers.BatchNormalization()) #Including batch normalization from Chapter 7 to attempt t
    model.add(layers.Dense(3000, activation='relu')) #This would be second layer and using the best s
    model.add(layers.BatchNormalization()) #Including batch normalization from Chapter 7 to attempt t
    model.add(layers.Dense(1, activation = 'linear'))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
model.summary()
```



```
First Layer Nodes =  10 ,  Second Layer Nodes =  10
43136/43136 [==============================] - 1s 22us/step
##############################################
The FL_Node:  10
The SL_Node:  10
('mean_squared_error', 940090.6292221114)
##############################################
First Layer Nodes =  10 ,  Second Layer Nodes =  20
43136/43136 [==============================] - 1s 22us/step
##############################################
The FL_Node:  10
The SL_Node:  20
('mean_squared_error', 843431.9070382047)
##############################################
First Layer Nodes =  10 ,  Second Layer Nodes =  30
43136/43136 [==============================] - 1s 23us/step
##############################################
The FL_Node:  10
The SL_Node:  30
('mean_squared_error', 800176.3266239336)
##############################################
First Layer Nodes =  10 ,  Second Layer Nodes =  40
43136/43136 [==============================] - 1s 23us/step
##############################################
The FL_Node:  10
The SL_Node:  40
('mean_squared_error', 757695.7939713928)
##############################################
First Layer Nodes =  10 ,  Second Layer Nodes =  50
43136/43136 [==============================] - 1s 23us/step
##############################################
The FL_Node:  10
The SL_Node:  50
('mean_squared_error', 756552.9769537046)
##############################################
```



```
##############################################
First Layer Nodes =  5000 ,  Second Layer Nodes =  2000
43136/43136 [==============================] - 2s 54us/step
##############################################
The FL_Node:  5000
The SL_Node:  2000
('mean_squared_error', 293144.2704484074)
##############################################
First Layer Nodes =  5000 ,  Second Layer Nodes =  2500
43136/43136 [==============================] - 2s 56us/step
##############################################
The FL_Node:  5000
The SL_Node:  2500
('mean_squared_error', 310477.0281203635)
##############################################
First Layer Nodes =  5000 ,  Second Layer Nodes =  3000
43136/43136 [==============================] - 3s 59us/step
##############################################
The FL_Node:  5000
The SL_Node:  3000
('mean_squared_error', 289876.00741984654)
##############################################
Time required for training:  0:26:38.291231
```

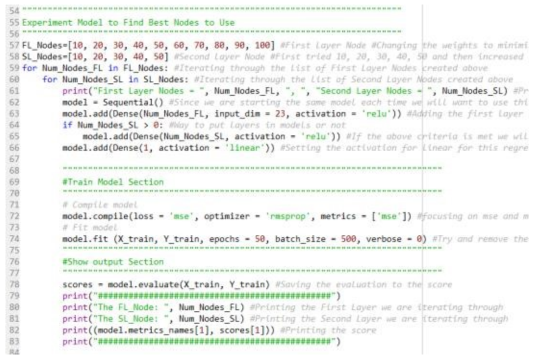


```
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Training The Final Model
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
model = build_model()
model.fit(X_train, Y_train,
          epochs = 50, batch_size = 500, verbose = 0)
test_mse_score, test_mae_score = model.evaluate(X_test, Y_test)

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Final Test MAE Score
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
print("##############################################")
print('Final Model Test MAE Score: ', test_mae_score)
print('Final Model Test MSE Score: ', test_mse_score)
print("##############################################")
```

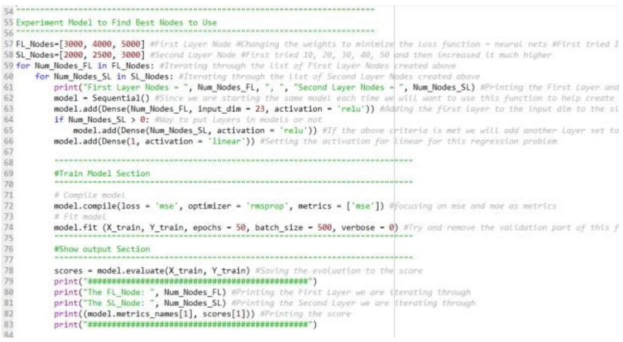
5. Your experimental plan for arriving at the final model.

For my experimental plan to arrive at my final model I used an experiment model to help find the best nodes that I should use later when I compile my final model. This experimentation consisted of searching through a list of first layer nodes I would set and then a list of second layer nodes and based on these iterations it would compile a test model setting the layers to the iteration value to later train and produce different mean squared errors. This was trained on the X_train and the Y_train that was created earlier and evaluated on MSE. I left the number of training epochs to 300 to give it plenty to train on and left the batch size to 500 for each iteration. I left those two variables as constants and mainly experimented with the first layer and second layer nodes to search through and combine for the lowest MSE.

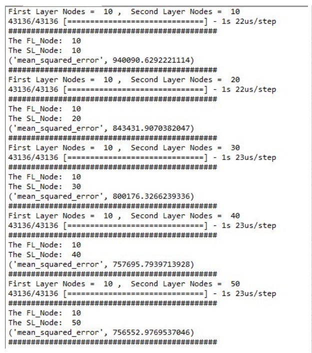
This is when I noticed that just using a list of first layer nodes from 10 to 100 and second layer 10 to 50 was not high enough due to my dataset and mainly resulted in still a high MSE. To lower this, I increased the first-layer nodes to test to 3000 to 5000 and second-layer nodes to 2000 to 3000. I tried going even higher but didn't find my MSE decreasing a significant amount around this threshold. Below I listed just the different lists I used for this experiment part, but I also listed how I would evaluate the scores to compare the MSE against one another. I provided all my results from my initial list, but if you look at them closely you'll see the MSE is still very large. My last screenshot shows my best results when my first layer node was 5000 and second layer was 3000 for a MSE of 289876.00. When running my final model, I just left the first layer to 5000 and second layer to 3000, since I was done experimenting.



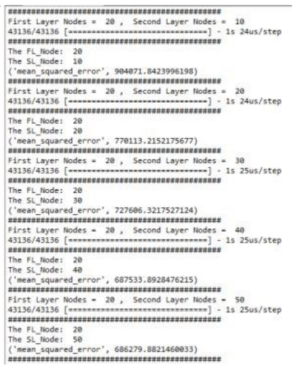
```
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Experiment Model to Find Best Nodes to Use
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
FL_Nodes=[10, 20, 30, 40, 50, 60, 70, 80, 90, 100] #First Layer Node #Changing the weights to minimi
SL_Nodes=[10, 20, 30, 40, 50] #Second Layer Node #First tried 10, 20, 30, 40, 50 and then increased
for Num_Nodes_FL in FL_Nodes: #Iterating through the list of First Layer Nodes created above
    for Num_Nodes_SL in SL_Nodes: #Iterating through the list of Second Layer Nodes created above
        print("First Layer Nodes = ", Num_Nodes_FL, ", ", "Second Layer Nodes = ", Num_Nodes_SL) #Pr
        model = Sequential() #Since we are starting the same model each time we will want to use thi
        model.add(Dense(Num_Nodes_FL, input_dim = 23, activation = 'relu')) #Adding the first layer
        if Num_Nodes_SL > 0: #Way to put layers in models or not
            model.add(Dense(Num_Nodes_SL, activation = 'relu')) #If the above criteria is met we wil
        model.add(Dense(1, activation = 'linear')) #Setting the activation for linear for this regre

        """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
        #Train Model Section
        """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
        # Compile model
        model.compile(loss = 'mse', optimizer = 'rmsprop', metrics = ['mse']) #focusing on mse and m
        # Fit model
        model.fit (X_train, Y_train, epochs = 50, batch_size = 500, verbose = 0) #Try and remove the
        """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
        #Show output Section
        """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
        scores = model.evaluate(X_train, Y_train) #Saving the evaluation to the score
        print("################################################")
        print("The FL_Node: ", Num_Nodes_FL) #Printing the First Layer we are iterating through
        print("The SL_Node: ", Num_Nodes_SL) #Printing the Second Layer we are iterating through
        print((model.metrics_names[1], scores[1])) #Printing the score
        print("################################################")
```



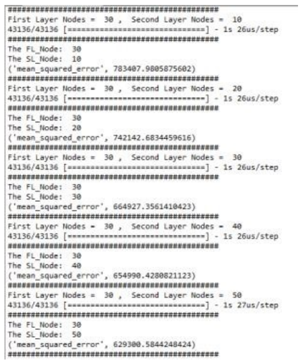
```
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Experiment Model to Find Best Nodes to Use
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
FL_Nodes=[3000, 4000, 5000] #First Layer Node #Changing the weights to minimize the loss function - neural nets #First tried 1
SL_Nodes=[2000, 2500, 3000] #Second Layer Node #First tried 10, 20, 30, 40, 50 and then increased it much higher
for Num_Nodes_FL in FL_Nodes: #Iterating through the list of First Layer Nodes created above
    for Num_Nodes_SL in SL_Nodes: #Iterating through the list of Second Layer Nodes created above
        print("First Layer Nodes = ", Num_Nodes_FL, ", ", "Second Layer Nodes = ", Num_Nodes_SL) #Printing the First Layer and
        model = Sequential() #Since we are starting the same model each time we will want to use this function to help create
        model.add(Dense(Num_Nodes_FL, input_dim = 23, activation = 'relu')) #Adding the first layer to the input dim to the si
        if Num_Nodes_SL > 0: #Way to put layers in models or not
            model.add(Dense(Num_Nodes_SL, activation = 'relu')) #If the above criteria is met we will add another layer set to
        model.add(Dense(1, activation = 'linear')) #Setting the activation for linear for this regression problem

        """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
        #Train Model Section
        """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
        # Compile model
        model.compile(loss = 'mse', optimizer = 'rmsprop', metrics = ['mse']) #focusing on mse and mae as metrics
        # Fit model
        model.fit (X_train, Y_train, epochs = 50, batch_size = 500, verbose = 0) #Try and remove the validation part of this f
        """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
        #Show output Section
        """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
        scores = model.evaluate(X_train, Y_train) #Saving the evaluation to the score
        print("################################################")
        print("The FL_Node: ", Num_Nodes_FL) #Printing the First Layer we are iterating through
        print("The SL_Node: ", Num_Nodes_SL) #Printing the Second Layer we are iterating through
        print((model.metrics_names[1], scores[1])) #Printing the score
        print("################################################")
```



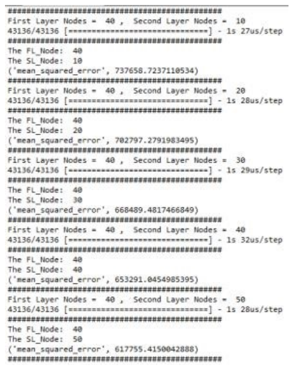
```
First Layer Nodes =  10 ,  Second Layer Nodes =  10
43136/43136 [==============================] - 1s 22us/step
################################################
The FL_Node:  10
The SL_Node:  10
('mean_squared_error', 940090.6292221114)
################################################
First Layer Nodes =  10 ,  Second Layer Nodes =  20
43136/43136 [==============================] - 1s 22us/step
################################################
The FL_Node:  10
The SL_Node:  20
('mean_squared_error', 843431.9070382047)
################################################
First Layer Nodes =  10 ,  Second Layer Nodes =  30
43136/43136 [==============================] - 1s 23us/step
################################################
The FL_Node:  10
The SL_Node:  30
('mean_squared_error', 800176.3266239336)
################################################
First Layer Nodes =  10 ,  Second Layer Nodes =  40
43136/43136 [==============================] - 1s 23us/step
################################################
The FL_Node:  10
The SL_Node:  40
('mean_squared_error', 757695.7939713928)
################################################
First Layer Nodes =  10 ,  Second Layer Nodes =  50
43136/43136 [==============================] - 1s 23us/step
################################################
The FL_Node:  10
The SL_Node:  50
('mean_squared_error', 756552.9769537046)
################################################
```



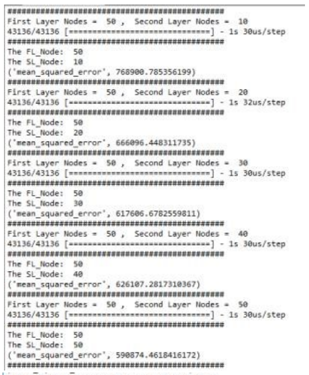
```
################################################
First Layer Nodes =  20 ,  Second Layer Nodes =  10
43136/43136 [==============================] - 1s 24us/step
################################################
The FL_Node:  20
The SL_Node:  10
('mean_squared_error', 904071.8423996198)
################################################
First Layer Nodes =  20 ,  Second Layer Nodes =  20
43136/43136 [==============================] - 1s 24us/step
################################################
The FL_Node:  20
The SL_Node:  20
('mean_squared_error', 770113.2152175677)
################################################
First Layer Nodes =  20 ,  Second Layer Nodes =  30
43136/43136 [==============================] - 1s 25us/step
################################################
The FL_Node:  20
The SL_Node:  30
('mean_squared_error', 727606.3217527124)
################################################
First Layer Nodes =  20 ,  Second Layer Nodes =  40
43136/43136 [==============================] - 1s 25us/step
################################################
The FL_Node:  20
The SL_Node:  40
('mean_squared_error', 687533.8928476215)
################################################
First Layer Nodes =  20 ,  Second Layer Nodes =  50
43136/43136 [==============================] - 1s 25us/step
################################################
The FL_Node:  20
The SL_Node:  50
('mean_squared_error', 686279.8821460033)
################################################
```



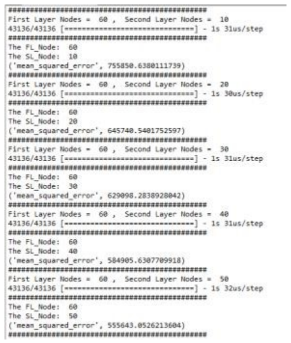
```
################################################
First Layer Nodes =  30 ,  Second Layer Nodes =  10
43136/43136 [==============================] - 1s 26us/step
################################################
The FL_Node:  30
The SL_Node:  10
('mean_squared_error', 783407.9805875602)
################################################
First Layer Nodes =  30 ,  Second Layer Nodes =  20
43136/43136 [==============================] - 1s 26us/step
################################################
The FL_Node:  30
The SL_Node:  20
('mean_squared_error', 742142.6834459616)
################################################
First Layer Nodes =  30 ,  Second Layer Nodes =  30
43136/43136 [==============================] - 1s 26us/step
################################################
The FL_Node:  30
The SL_Node:  30
('mean_squared_error', 664927.3561410423)
################################################
First Layer Nodes =  30 ,  Second Layer Nodes =  40
43136/43136 [==============================] - 1s 26us/step
################################################
The FL_Node:  30
The SL_Node:  40
('mean_squared_error', 654990.4280821123)
################################################
First Layer Nodes =  30 ,  Second Layer Nodes =  50
43136/43136 [==============================] - 1s 27us/step
################################################
The FL_Node:  30
The SL_Node:  50
('mean_squared_error', 629300.5844248424)
################################################
```



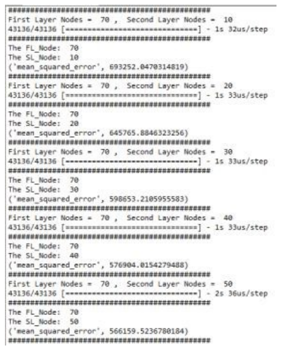
```
################################################
First Layer Nodes =  40 ,  Second Layer Nodes =  10
43136/43136 [==============================] - 1s 27us/step
################################################
The FL_Node:  40
The SL_Node:  10
('mean_squared_error', 737658.7237110534)
################################################
First Layer Nodes =  40 ,  Second Layer Nodes =  20
43136/43136 [==============================] - 1s 28us/step
################################################
The FL_Node:  40
The SL_Node:  20
('mean_squared_error', 702797.2791983495)
################################################
First Layer Nodes =  40 ,  Second Layer Nodes =  30
43136/43136 [==============================] - 1s 29us/step
################################################
The FL_Node:  40
The SL_Node:  30
('mean_squared_error', 668489.4817466849)
################################################
First Layer Nodes =  40 ,  Second Layer Nodes =  40
43136/43136 [==============================] - 1s 32us/step
################################################
The FL_Node:  40
The SL_Node:  40
('mean_squared_error', 653291.0454985395)
################################################
First Layer Nodes =  40 ,  Second Layer Nodes =  50
43136/43136 [==============================] - 1s 28us/step
################################################
The FL_Node:  40
The SL_Node:  50
('mean_squared_error', 617755.4150042888)
################################################
```



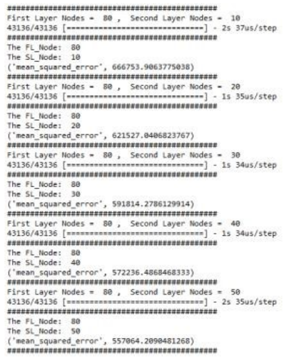
```
################################################
First Layer Nodes =  50 ,  Second Layer Nodes =  10
43136/43136 [==============================] - 1s 30us/step
################################################
The FL_Node:  50
The SL_Node:  10
('mean_squared_error', 768900.785356199)
################################################
First Layer Nodes =  50 ,  Second Layer Nodes =  20
43136/43136 [==============================] - 1s 32us/step
################################################
The FL_Node:  50
The SL_Node:  20
('mean_squared_error', 666096.448311735)
################################################
First Layer Nodes =  50 ,  Second Layer Nodes =  30
43136/43136 [==============================] - 1s 30us/step
################################################
The FL_Node:  50
The SL_Node:  30
('mean_squared_error', 617606.6782559811)
################################################
First Layer Nodes =  50 ,  Second Layer Nodes =  40
43136/43136 [==============================] - 1s 30us/step
################################################
The FL_Node:  50
The SL_Node:  40
('mean_squared_error', 626107.2817310367)
################################################
First Layer Nodes =  50 ,  Second Layer Nodes =  50
43136/43136 [==============================] - 1s 30us/step
################################################
The FL_Node:  50
The SL_Node:  50
('mean_squared_error', 590874.4618416172)
################################################
```



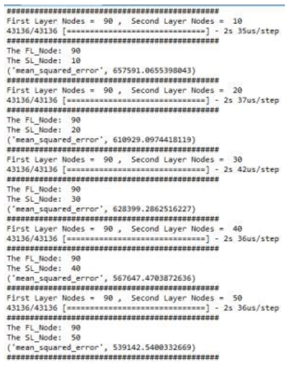
```
################################################
First Layer Nodes =  60 ,  Second Layer Nodes =  10
43136/43136 [==============================] - 1s 31us/step
################################################
The FL_Node:  60
The SL_Node:  10
('mean_squared_error', 755850.6380111739)
################################################
First Layer Nodes =  60 ,  Second Layer Nodes =  20
43136/43136 [==============================] - 1s 30us/step
################################################
The FL_Node:  60
The SL_Node:  20
('mean_squared_error', 645740.5401752597)
################################################
First Layer Nodes =  60 ,  Second Layer Nodes =  30
43136/43136 [==============================] - 1s 31us/step
################################################
The FL_Node:  60
The SL_Node:  30
('mean_squared_error', 629098.2838928042)
################################################
First Layer Nodes =  60 ,  Second Layer Nodes =  40
43136/43136 [==============================] - 1s 31us/step
################################################
The FL_Node:  60
The SL_Node:  40
('mean_squared_error', 584905.6307709918)
################################################
First Layer Nodes =  60 ,  Second Layer Nodes =  50
43136/43136 [==============================] - 1s 32us/step
################################################
The FL_Node:  60
The SL_Node:  50
('mean_squared_error', 555643.0526213604)
################################################
```



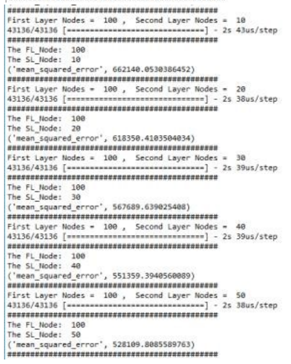
```
################################################
First Layer Nodes =  70 ,  Second Layer Nodes =  10
43136/43136 [==============================] - 1s 32us/step
################################################
The FL_Node:  70
The SL_Node:  10
('mean_squared_error', 693252.0470314819)
################################################
First Layer Nodes =  70 ,  Second Layer Nodes =  20
43136/43136 [==============================] - 1s 33us/step
################################################
The FL_Node:  70
The SL_Node:  20
('mean_squared_error', 645765.8846323256)
################################################
First Layer Nodes =  70 ,  Second Layer Nodes =  30
43136/43136 [==============================] - 1s 33us/step
################################################
The FL_Node:  70
The SL_Node:  30
('mean_squared_error', 598653.2105955583)
################################################
First Layer Nodes =  70 ,  Second Layer Nodes =  40
43136/43136 [==============================] - 1s 33us/step
################################################
The FL_Node:  70
The SL_Node:  40
('mean_squared_error', 576904.0154279488)
################################################
First Layer Nodes =  70 ,  Second Layer Nodes =  50
43136/43136 [==============================] - 2s 36us/step
################################################
The FL_Node:  70
The SL_Node:  50
('mean_squared_error', 566159.5236780184)
################################################
```



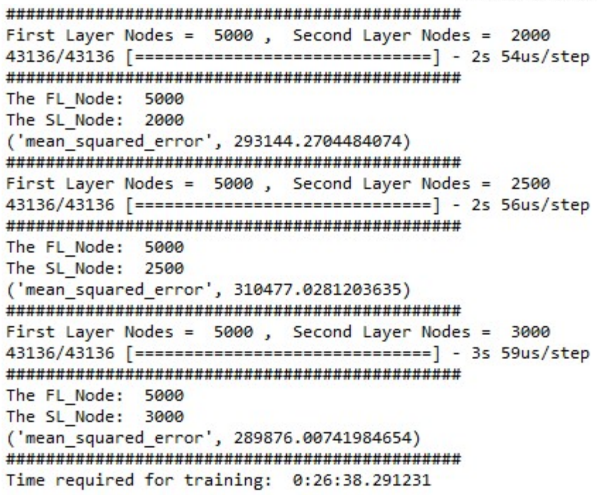
```
################################################
First Layer Nodes =  80 ,  Second Layer Nodes =  10
43136/43136 [==============================] - 2s 37us/step
################################################
The FL_Node:  80
The SL_Node:  10
('mean_squared_error', 666753.9063775038)
################################################
First Layer Nodes =  80 ,  Second Layer Nodes =  20
43136/43136 [==============================] - 1s 35us/step
################################################
The FL_Node:  80
The SL_Node:  20
('mean_squared_error', 621527.0406823767)
################################################
First Layer Nodes =  80 ,  Second Layer Nodes =  30
43136/43136 [==============================] - 1s 34us/step
################################################
The FL_Node:  80
The SL_Node:  30
('mean_squared_error', 591814.2786129914)
################################################
First Layer Nodes =  80 ,  Second Layer Nodes =  40
43136/43136 [==============================] - 1s 34us/step
################################################
The FL_Node:  80
The SL_Node:  40
('mean_squared_error', 572236.4868468333)
################################################
First Layer Nodes =  80 ,  Second Layer Nodes =  50
43136/43136 [==============================] - 2s 35us/step
################################################
The FL_Node:  80
The SL_Node:  50
('mean_squared_error', 557064.2090481268)
################################################
```



```
################################################
First Layer Nodes =  90 ,  Second Layer Nodes =  10
43136/43136 [==============================] - 2s 35us/step
################################################
The FL_Node:  90
The SL_Node:  10
('mean_squared_error', 657591.0655398043)
################################################
First Layer Nodes =  90 ,  Second Layer Nodes =  20
43136/43136 [==============================] - 2s 37us/step
################################################
The FL_Node:  90
The SL_Node:  20
('mean_squared_error', 610929.0974418119)
################################################
First Layer Nodes =  90 ,  Second Layer Nodes =  30
43136/43136 [==============================] - 2s 42us/step
################################################
The FL_Node:  90
The SL_Node:  30
('mean_squared_error', 628399.2862516227)
################################################
First Layer Nodes =  90 ,  Second Layer Nodes =  40
43136/43136 [==============================] - 2s 36us/step
################################################
The FL_Node:  90
The SL_Node:  40
('mean_squared_error', 567647.4703872636)
################################################
First Layer Nodes =  90 ,  Second Layer Nodes =  50
43136/43136 [==============================] - 2s 36us/step
################################################
The FL_Node:  90
The SL_Node:  50
('mean_squared_error', 539142.5400332669)
################################################
```



```
################################################
First Layer Nodes =  100 ,  Second Layer Nodes =  10
43136/43136 [==============================] - 2s 43us/step
################################################
The FL_Node:  100
The SL_Node:  10
('mean_squared_error', 662140.0530386452)
################################################
First Layer Nodes =  100 ,  Second Layer Nodes =  20
43136/43136 [==============================] - 2s 38us/step
################################################
The FL_Node:  100
The SL_Node:  20
('mean_squared_error', 618350.4103504034)
################################################
First Layer Nodes =  100 ,  Second Layer Nodes =  30
43136/43136 [==============================] - 2s 39us/step
################################################
The FL_Node:  100
The SL_Node:  30
('mean_squared_error', 567689.639025408)
################################################
First Layer Nodes =  100 ,  Second Layer Nodes =  40
43136/43136 [==============================] - 2s 39us/step
################################################
The FL_Node:  100
The SL_Node:  40
('mean_squared_error', 551359.3940560089)
################################################
First Layer Nodes =  100 ,  Second Layer Nodes =  50
43136/43136 [==============================] - 2s 38us/step
################################################
The FL_Node:  100
The SL_Node:  50
('mean_squared_error', 528109.8085589763)
################################################
```

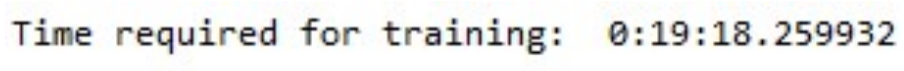


```
################################################
First Layer Nodes =  5000 ,  Second Layer Nodes =  2000
43136/43136 [==============================] - 2s 54us/step
################################################
The FL_Node:  5000
The SL_Node:  2000
('mean_squared_error', 293144.2704484074)
################################################
First Layer Nodes =  5000 ,  Second Layer Nodes =  2500
43136/43136 [==============================] - 2s 56us/step
################################################
The FL_Node:  5000
The SL_Node:  2500
('mean_squared_error', 310477.0281203635)
################################################
First Layer Nodes =  5000 ,  Second Layer Nodes =  3000
43136/43136 [==============================] - 3s 59us/step
################################################
The FL_Node:  5000
The SL_Node:  3000
('mean_squared_error', 289876.00741984654)
################################################
Time required for training:  0:26:38.291231
```

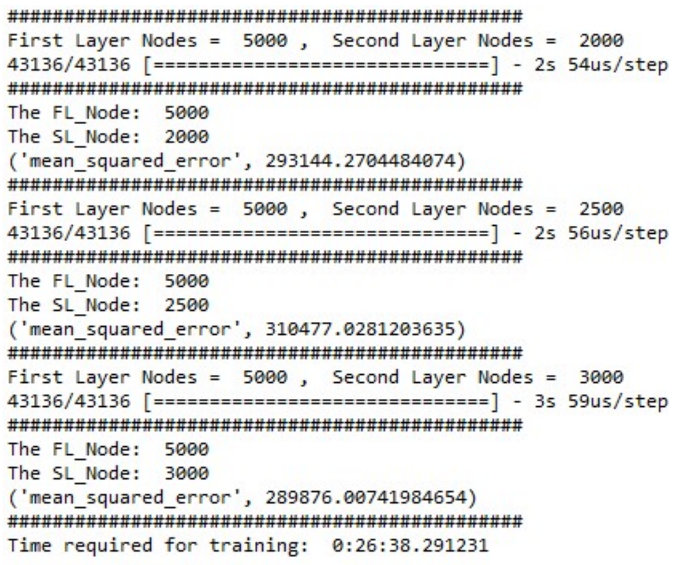
6. How long it took to run all the models in your experimental plan.

   On average while experimenting initially with my list of first-layer nodes 10 to 100 and second-layer nodes of 10 to 50 the time it took to run all my models was 19 minutes and 18 seconds. However, when I increased my list size of first-layer nodes to 3000 to 5000 and second-layer nodes to 2000 to 3000 the time to train this model took 26 minutes and 38 seconds.

The number of epochs also significantly impacted this process and as a rule of thumb I thought about using 200 epochs but wanted to see how 300 epochs would help. We can view later when I begin plotting that there does seem to be some overfitting, but I wanted to view this on the plots. I also believe this increased time overall for this experimenting model to run is due to the large dataset that I am working with as well. I did test these multiple times with increasing the batch size as well and though the time did increase slightly the overall MSE that I was trying to lower do not significantly change so I just decided to keep a constant batch size of 500 for the rest of my experimentation. When I was ready to run everything for my model after experimentation it took my model a total of 4 hours to complete.



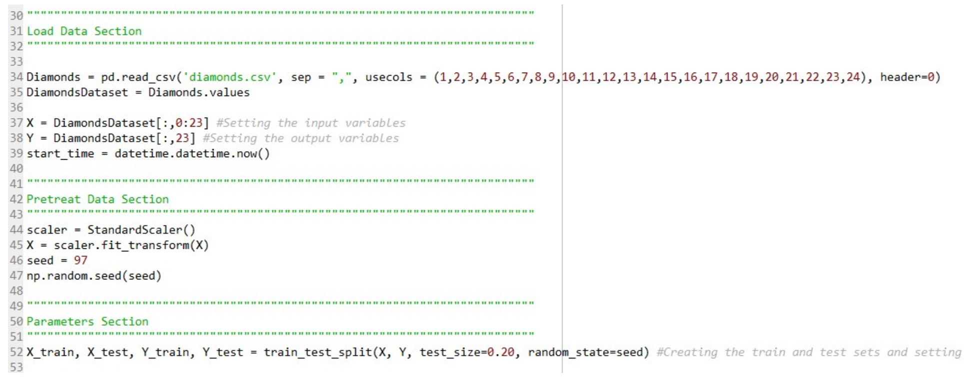
```
Time required for training:  0:19:18.259932
```



```
################################################
First Layer Nodes =  5000 ,  Second Layer Nodes =  2000
43136/43136 [==============================] - 2s 54us/step
################################################
The FL_Node:  5000
The SL_Node:  2000
('mean_squared_error', 293144.2704484074)
################################################
First Layer Nodes =  5000 ,  Second Layer Nodes =  2500
43136/43136 [==============================] - 2s 56us/step
################################################
The FL_Node:  5000
The SL_Node:  2500
('mean_squared_error', 310477.0281203635)
################################################
First Layer Nodes =  5000 ,  Second Layer Nodes =  3000
43136/43136 [==============================] - 3s 59us/step
################################################
The FL_Node:  5000
The SL_Node:  3000
('mean_squared_error', 289876.00741984654)
################################################
Time required for training:  0:26:38.291231
```

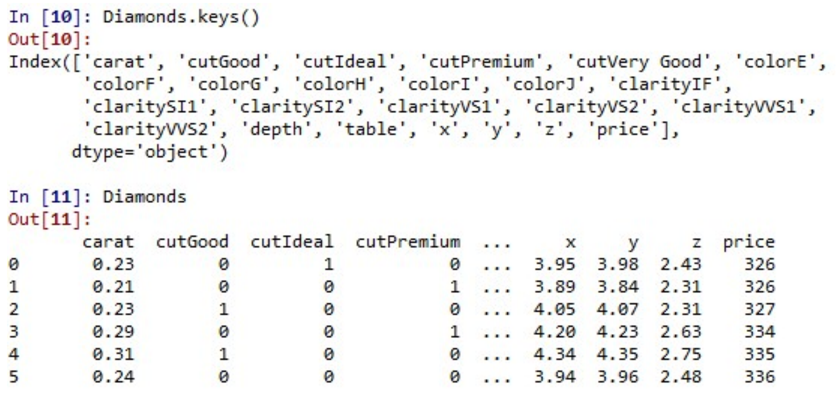
7. An explanation of the input variables

My Input variables for my project were first read in into my model and saved as the variable name Diamonds. Since, my dataset already had a header name for each of the columns I was able to just read everything in and made sure to identify which columns I wanted to use as well, which was all of them except the first column that contained row ID. I then set my data values to a dataset variable named DiamondsDataset. All my input variables from the Diamonds Kaggle dataset are attributes that help contribute towards pricing diamonds. Each row is a different diamond with its own unique carat size, cut, color, clarity, depth, table, x (length), y (width), z (depth), and price. Since three of the input variables cut, color, and clarity are qualitative variables I had set them to dummy variables in RStudio before saving the new csv file and reading that in. I then set X to the first 23 columns of the dataset since these are my input variables or dependent variables and then set Y to the final column price since this is my output variable or independent variable. I then pretreated the data by scaling the data and used this data to create my X_train, Y_train, X_test, and Y_test using an 80 / 20 split, but I will discuss more pretreating steps in the next question. You can view the keys or input variables of my dataset below as well as the first five example rows from the csv file.



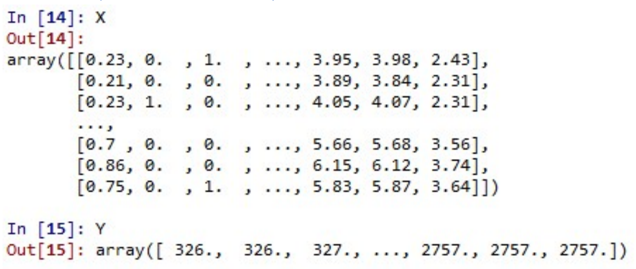
```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Load Data Section
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

Diamonds = pd.read_csv('diamonds.csv', sep = ",", usecols = (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24), header=0)
DiamondsDataset = Diamonds.values

X = DiamondsDataset[:,0:23] #Setting the input variables
Y = DiamondsDataset[:,23] #Setting the output variables
start_time = datetime.datetime.now()

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Pretreat Data Section
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
scaler = StandardScaler()
X = scaler.fit_transform(X)
seed = 97
np.random.seed(seed)

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Parameters Section
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.20, random_state=seed) #Creating the train and test sets and setting

```

My dataset keys and first five example rows



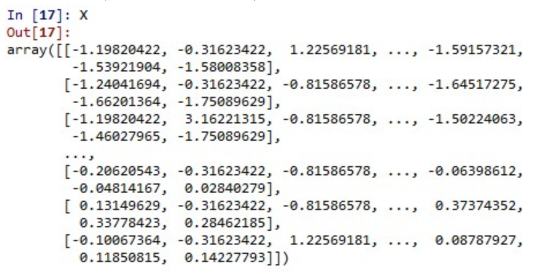
```
In [10]: Diamonds.keys()
Out[10]:
Index(['carat', 'cutGood', 'cutIdeal', 'cutPremium', 'cutVery Good', 'colorE',
       'colorF', 'colorG', 'colorH', 'colorI', 'colorJ', 'clarityIF',
       'claritySI1', 'claritySI2', 'clarityVS1', 'clarityVS2', 'clarityVVS1',
       'clarityVVS2', 'depth', 'table', 'x', 'y', 'z', 'price'],
      dtype='object')

In [11]: Diamonds
Out[11]:
   carat  cutGood  cutIdeal  cutPremium  ...     x     y     z  price
0   0.23        0         1           0  ...  3.95  3.98  2.43    326
1   0.21        0         0           1  ...  3.89  3.84  2.31    326
2   0.23        1         0           0  ...  4.05  4.07  2.31    327
3   0.29        0         0           1  ...  4.20  4.23  2.63    334
4   0.31        1         0           0  ...  4.34  4.35  2.75    335
5   0.24        0         0           0  ...  3.94  3.96  2.48    336
```

X (before scaled)



```
In [14]: X
Out[14]:
array([[0.23, 0.  , 1.  , ..., 3.95, 3.98, 2.43],
       [0.21, 0.  , 0.  , ..., 3.89, 3.84, 2.31],
       [0.23, 1.  , 0.  , ..., 4.05, 4.07, 2.31],
       ...,
       [0.7 , 0.  , 0.  , ..., 5.66, 5.68, 3.56],
       [0.86, 0.  , 0.  , ..., 6.15, 6.12, 3.74],
       [0.75, 0.  , 1.  , ..., 5.83, 5.87, 3.64]])

In [15]: Y
Out[15]: array([ 326.,  326.,  327., ..., 2757., 2757., 2757.])
```

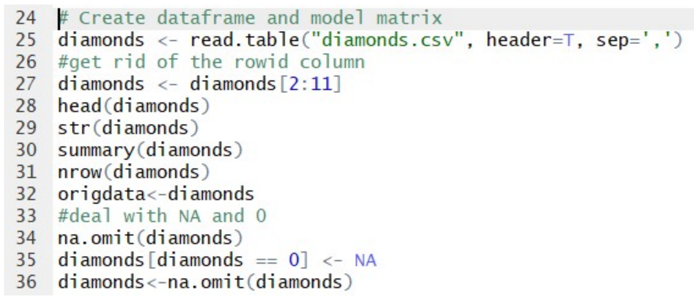
X (after scaled)



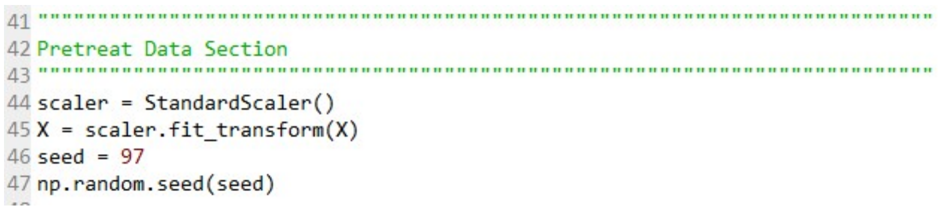
```
In [17]: X
Out[17]:
array([[-1.19820422, -0.31623422,  1.22569181, ..., -1.59157321,
        -1.53921904, -1.58008358],
       [-1.24041694, -0.31623422, -0.81586578, ..., -1.64517275,
        -1.66201364, -1.75089629],
       [-1.19820422,  3.16221315, -0.81586578, ..., -1.50224063,
        -1.46027965, -1.75089629],
       ...,
       [-0.20620543, -0.31623422, -0.81586578, ..., -0.06398612,
        -0.04814167,  0.02840279],
       [ 0.13149629, -0.31623422, -0.81586578, ...,  0.37374352,
         0.33778423,  0.28462185],
       [-0.10067364, -0.31623422,  1.22569181, ...,  0.08787927,
         0.11850815,  0.14227793]])
```

8. The data preprocessing steps you took. And why you took those steps.

   For the data preprocessing steps, I was fortunate that my selected dataset was a relatively clean dataset from the beginning. I explored the data in RStudio and noticed that I did not have any missing values (NA), but I did have 0s in the length, width, and depth columns. Since, there are no such things as a 2D diamond and we should only be working with 3D objects, I decided to set those 0s to NAs and then removed them. Below I have provided just a quick screen shot of this process. There were only 20 rows that had this issue, so I justified that it was okay dropping 20 rows out of the almost 54,000 rows I had. Due to this being a Kaggle dataset as well I assumed that those 0s might have been placed purposely to make sure you do remove them. Once all this data preprocessing was taken care of during the pretreat data section of my model I went ahead and scaled my x values and set a seed to 97 (My favorite Ice Hockey number!).

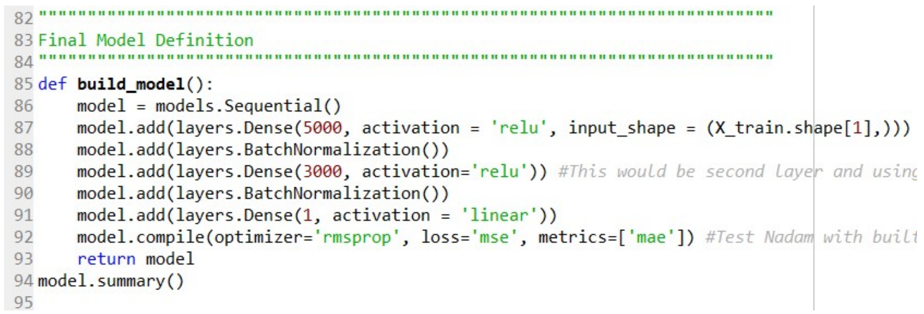


```
# Create dataframe and model matrix
diamonds <- read.table("diamonds.csv", header=T, sep=',')
#get rid of the rowid column
diamonds <- diamonds[2:11]
head(diamonds)
str(diamonds)
summary(diamonds)
nrow(diamonds)
origdata<-diamonds
#deal with NA and 0
na.omit(diamonds)
diamonds[diamonds == 0] <- NA
diamonds<-na.omit(diamonds)
```



```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Pretreat Data Section
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
scaler = StandardScaler()
X = scaler.fit_transform(X)
seed = 97
np.random.seed(seed)
```

9. An explanation of your metrics and justification for your choice. For my final project I decided that my metrics I'd focus on would be MSE and MAE. I
decided to focus on the MSE loss function, since that would help with determining the square of the difference between the predictions and the targets and commonly used in regression problems. Focusing on the MAE as well would allow me to find the absolute value of the difference between the predictions and target of how far off the input variables are on average to help determine a better price predictor for our diamonds. Overall, since the loss function that is commonly associated with regression is MSE and MAE, I concluded that they would be good choices that can be used as the evaluation metrics. I was curious to see the lowest MAE I could achieve to know how far off our predictor input variables are on average when determining the price for diamonds.



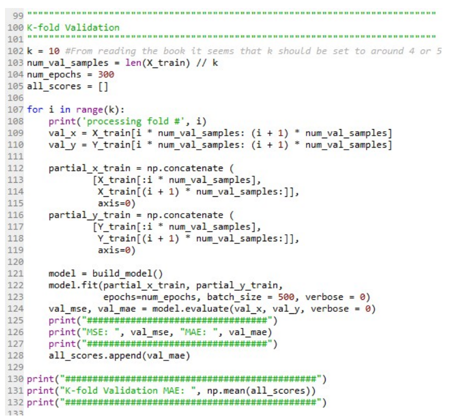
```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Final Model Definition
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
def build_model():
    model = models.Sequential()
    model.add(layers.Dense(5000, activation = 'relu', input_shape = (X_train.shape[1],)))
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(3000, activation='relu')) #This would be second layer and using
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(1, activation = 'linear'))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae']) #Test Nadam with buil
    return model
model.summary()
```

10.

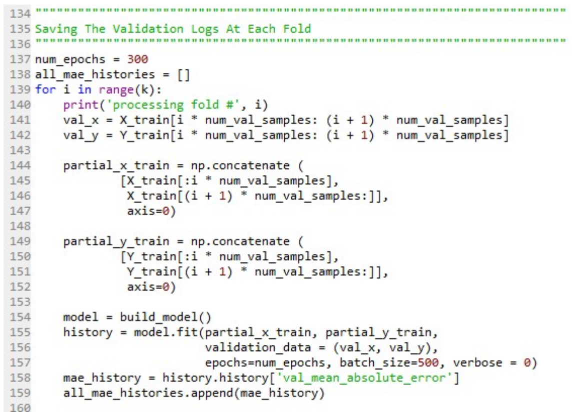
An explanation of your method to validate the model.

To validate this regression model, I focused on creating a K-fold validation process by utilizing the data and creating a training and validation set to use in this process. I wanted to attempt K-fold validation to take in consideration the high variance that I could potentially have in my dataset. I decided to attempt 10 K-Folds to test the different runs and check the different MAE results I would get with each iteration. In the K-fold validation for each of the iterations I would create a validation x and a validation y based on the X_train and Y_train that was multiplying the samples of the length of X_train by the K iteration. I also created a partial_x_train and partial_y_train that was also multiplying the K iteration by the length of the X_train divided by total k. Essentially all of these needed to be built for us to later train the partial variables against the validation dataset to determine the best K MAE we can get. Like my other models I also left the batch size at 500 and the number of epochs to 300 for this process. From the different folds the best I found was K fold #6 which resulted in a MAE of 283.02 or potentially off by $283.02 on average. Once this K-fold validation model was done I then saved the validation logs at each fold into a history variable containing the validation mean absolute errors. From this point I focused on plotting the validation scores and from the chart below we can see that this model is significantly lowering the validation MAE, which is a good sign for my validation, however past around 40 epochs or so we can begin to see that we are essentially just overfitting this model each time with the epoch iterations.

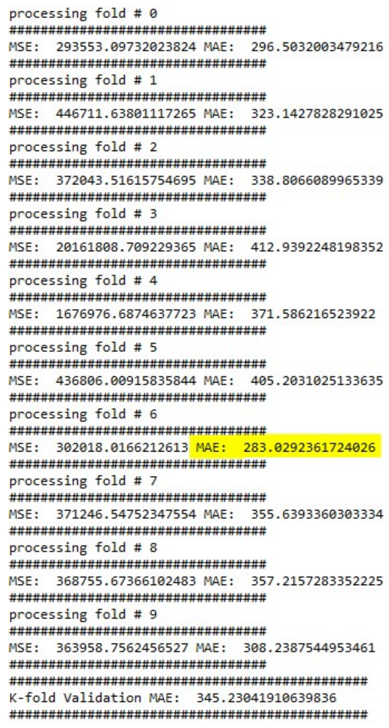
11.



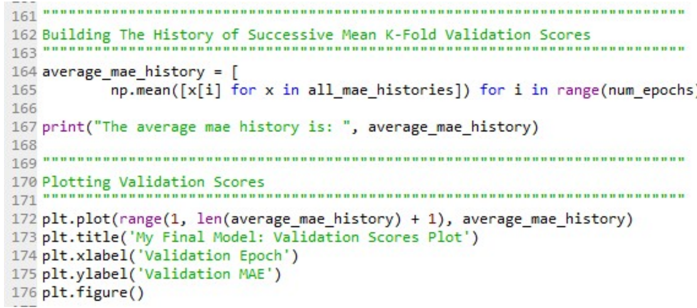
```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
K-fold Validation
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
k = 10 #From reading the book it seems that k should be set to around 4 or 5
num_val_samples = len(X_train) // k
num_epochs = 300
all_scores = []

for i in range(k):
    print('processing fold #', i)
    val_x = X_train[i * num_val_samples: (i + 1) * num_val_samples]
    val_y = Y_train[i * num_val_samples: (i + 1) * num_val_samples]

    partial_x_train = np.concatenate (
            [X_train[:i * num_val_samples],
             X_train[(i + 1) * num_val_samples:]],
             axis=0)
    partial_y_train = np.concatenate (
            [Y_train[:i * num_val_samples],
             Y_train[(i + 1) * num_val_samples:]],
             axis=0)

    model = build_model()
    model.fit(partial_x_train, partial_y_train,
              epochs=num_epochs, batch_size = 500, verbose = 0)
    val_mse, val_mae = model.evaluate(val_x, val_y, verbose = 0)
    print("##################################")
    print("MSE: ", val_mse, "MAE: ", val_mae)
    print("##################################")
    all_scores.append(val_mae)

print("##############################################")
print("K-fold Validation MAE: ", np.mean(all_scores))
print("##############################################")
```



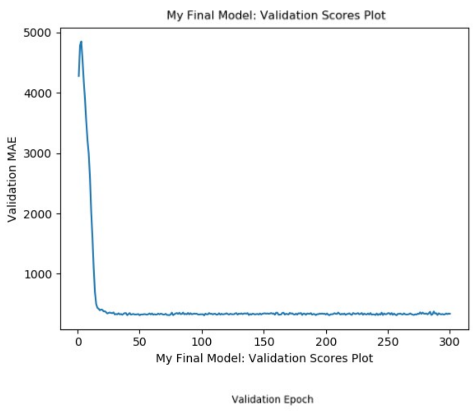
```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Saving The Validation Logs At Each Fold
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
num_epochs = 300
all_mae_histories = []
for i in range(k):
    print('processing fold #', i)
    val_x = X_train[i * num_val_samples: (i + 1) * num_val_samples]
    val_y = Y_train[i * num_val_samples: (i + 1) * num_val_samples]

    partial_x_train = np.concatenate (
            [X_train[:i * num_val_samples],
             X_train[(i + 1) * num_val_samples:]],
             axis=0)

    partial_y_train = np.concatenate (
            [Y_train[:i * num_val_samples],
             Y_train[(i + 1) * num_val_samples:]],
             axis=0)

    model = build_model()
    history = model.fit(partial_x_train, partial_y_train,
                        validation_data = (val_x, val_y),
                        epochs=num_epochs, batch_size=500, verbose = 0)
    mae_history = history.history['val_mean_absolute_error']
    all_mae_histories.append(mae_history)
```



```
processing fold # 0
##################################
MSE:  293553.09732023824 MAE:  296.5032003479216
##################################
processing fold # 1
##################################
MSE:  446711.63801117265 MAE:  323.1427828291025
##################################
processing fold # 2
##################################
MSE:  372043.51615754695 MAE:  338.8066089965339
##################################
processing fold # 3
##################################
MSE:  20161808.709229365 MAE:  412.9392248198352
##################################
processing fold # 4
##################################
MSE:  1676976.6874637723 MAE:  371.586216523922
##################################
processing fold # 5
##################################
MSE:  436806.00915835844 MAE:  405.2031025133635
##################################
processing fold # 6
##################################
MSE:  302018.0166212613 MAE:  283.0292361724026
##################################
processing fold # 7
##################################
MSE:  371246.54752347554 MAE:  355.6393360303334
##################################
processing fold # 8
##################################
MSE:  368755.67366102483 MAE:  357.2157283352225
##################################
processing fold # 9
##################################
MSE:  363958.7562456527 MAE:  308.2387544953461
##################################
##############################################
K-fold Validation MAE:  345.23041910639836
##############################################
```



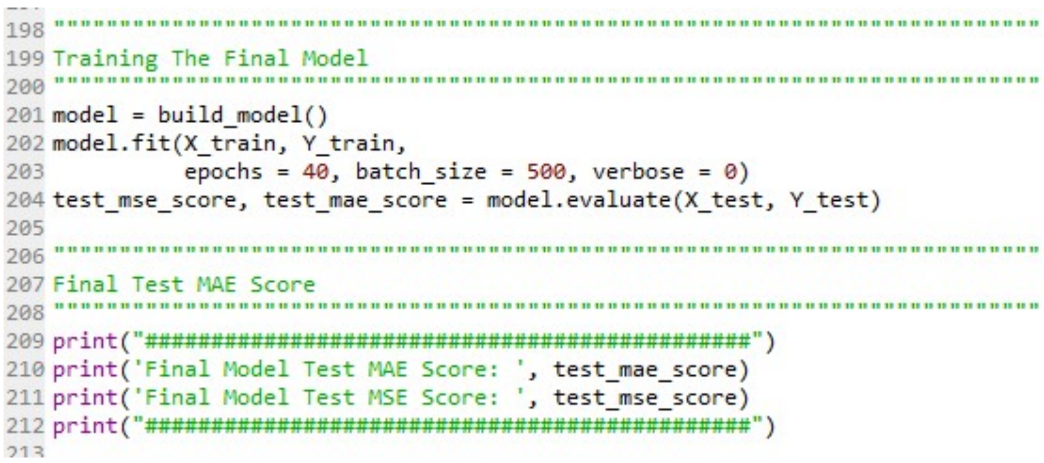
```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Building The History of Successive Mean K-Fold Validation Scores
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
average_mae_history = [
        np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]

print("The average mae history is: ", average_mae_history)

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Plotting Validation Scores
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
plt.plot(range(1, len(average_mae_history) + 1), average_mae_history)
plt.title('My Final Model: Validation Scores Plot')
plt.xlabel('Validation Epoch')
plt.ylabel('Validation MAE')
plt.figure()
```



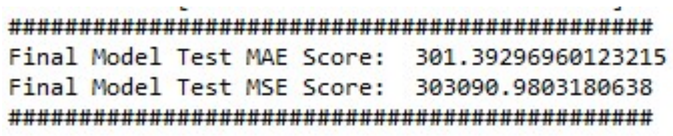
Your results in terms of appropriate metrics for the objective and problem.

Once all my previous sections beforehand completed running I trained a final model that was built based on fitting my X_train and Y_train I created earlier against the entire dataset. However, this time based on my validation step, since I determined that I was essentially overfitting my model after around 50 epochs I set this final model to run on 40 epochs to try and get the most accurate MAE of the entire dataset from the experimentation. After the model was done training I evaluated it against my X_test and Y_test dataset created earlier into final test_mse_score and test_mae_score variables. From my final MAE result of 301.39 I can determine that my model is still off around $301.39 dollars when it comes to predicting the price of diamonds based on the input variables I am using. Overall this isn't a terrible amount to be off around when predicting diamonds in the high price range, however I still would like to lower this even more.

12.



```
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Training The Final Model
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
model = build_model()
model.fit(X_train, Y_train,
          epochs = 40, batch_size = 500, verbose = 0)
test_mse_score, test_mae_score = model.evaluate(X_test, Y_test)

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Final Test MAE Score
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
print("################################################")
print('Final Model Test MAE Score: ', test_mae_score)
print('Final Model Test MSE Score: ', test_mse_score)
print("################################################")
```



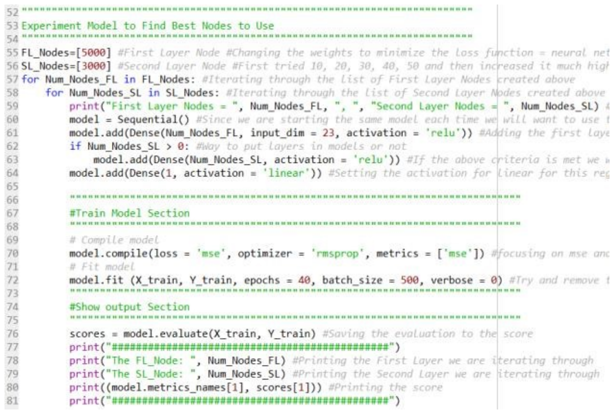
```
################################################
Final Model Test MAE Score:  301.39296960123215
Final Model Test MSE Score:  303090.9803180638
################################################
```

A discussion and/or justification for how you used/didn't use all the following:

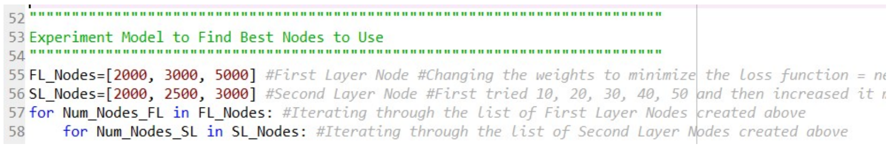
1. Selection of the optimum number of units

   For the optimal number of units to use for my model layers and dense this is when I focused on experimenting through the different nodes list to find the best combination that I could use to determine the lowest MSE. Overall, this took me a good number of hours to try different combinations trying to find the right units for each of the layers. The best number of units I found before it seemed my additional MSE score wasn't really decreasing significantly anymore was using 5000 for my first layer units and 3000 for my second layer units. Below are some of the experiment trails that I attempted to help determine the number of units I wanted to use.

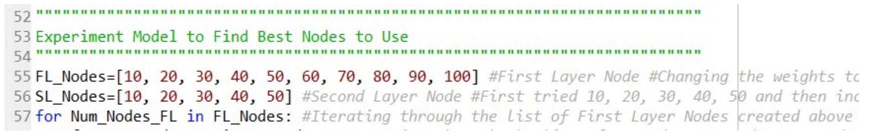
13.



```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Experiment Model to Find Best Nodes to Use
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
FL_Nodes=[5000] #First Layer Node #Changing the weights to minimize the loss function = neural net
SL_Nodes=[3000] #Second Layer Node #First tried 10, 20, 30, 40, 50 and then increased it much high
for Num_Nodes_FL in FL_Nodes: #Iterating through the list of First Layer Nodes created above
    for Num_Nodes_SL in SL_Nodes: #Iterating through the list of Second Layer Nodes created above
        print("First Layer Nodes = ", Num_Nodes_FL, ", ", "Second Layer Nodes = ", Num_Nodes_SL) #
        model = Sequential() #Since we are starting the same model each time we will want to use t
        model.add(Dense(Num_Nodes_FL, input_dim = 23, activation = 'relu')) #Adding the first laye
        if Num_Nodes_SL > 0: #Way to put layers in models or not
            model.add(Dense(Num_Nodes_SL, activation = 'relu')) #If the above criteria is met we w
        model.add(Dense(1, activation = 'linear')) #Setting the activation for linear for this reg

        """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
        #Train Model Section
        """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

        # Compile model
        model.compile(loss = 'mse', optimizer = 'rmsprop', metrics = ['mse']) #focusing on mse anc
        # Fit model
        model.fit (X_train, Y_train, epochs = 40, batch_size = 500, verbose = 0) #Try and remove t
        """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
        #Show output Section
        """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

        scores = model.evaluate(X_train, Y_train) #Saving the evaluation to the score
        print("###############################################")
        print("The FL_Node: ", Num_Nodes_FL) #Printing the First Layer we are iterating through
        print("The SL_Node: ", Num_Nodes_SL) #Printing the Second Layer we are iterating through
        print((model.metrics_names[1], scores[1])) #Printing the score
        print("###############################################")
```



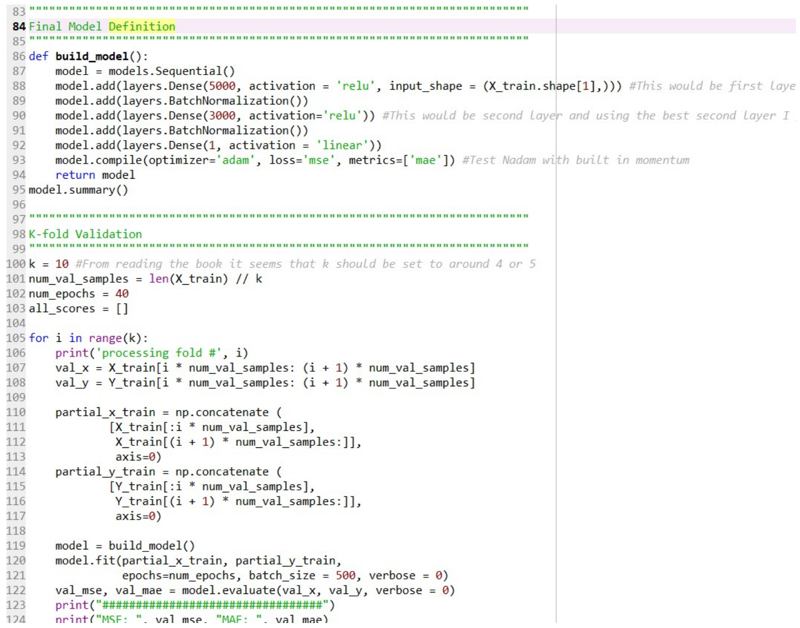
```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Experiment Model to Find Best Nodes to Use
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
FL_Nodes=[2000, 3000, 5000] #First Layer Node #Changing the weights to minimize the loss function = ne
SL_Nodes=[2000, 2500, 3000] #Second Layer Node #First tried 10, 20, 30, 40, 50 and then increased it r
for Num_Nodes_FL in FL_Nodes: #Iterating through the list of First Layer Nodes created above
    for Num_Nodes_SL in SL_Nodes: #Iterating through the list of Second Layer Nodes created above
```



```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Experiment Model to Find Best Nodes to Use
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
FL_Nodes=[10, 20, 30, 40, 50, 60, 70, 80, 90, 100] #First Layer Node #Changing the weights tc
SL_Nodes=[10, 20, 30, 40, 50] #Second Layer Node #First tried 10, 20, 30, 40, 50 and then inc
for Num_Nodes_FL in FL_Nodes: #Iterating through the list of First Layer Nodes created above
```

2. Type of network (feedforward, recurrent, backpropagation, and etc.),

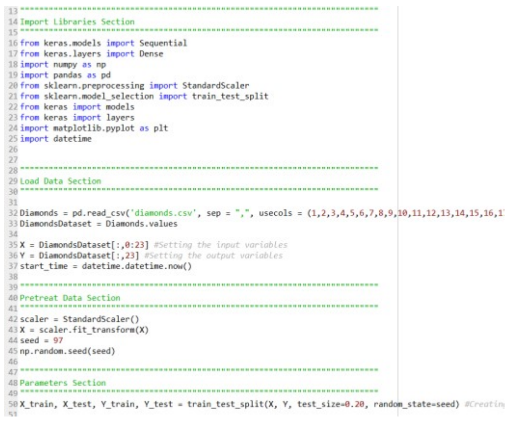
For my model it was created as a feedforward type of network. This is due to my model being fed the data and sequence independently and being run all at once. Since this is feedforward I am consistently feeding my model information at each step of my model to train it over time.



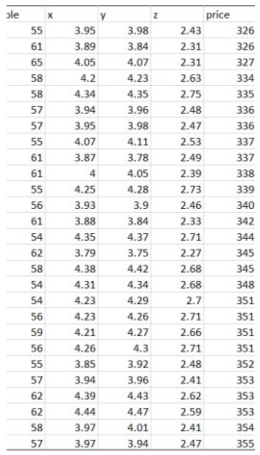
```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Final Model Definition
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
def build_model():
    model = models.Sequential()
    model.add(layers.Dense(5000, activation = 'relu', input_shape = (X_train.shape[1],))) #This would be first laye
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(3000, activation='relu')) #This would be second layer and using the best second layer I
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(1, activation = 'linear'))
    model.compile(optimizer='adam', loss='mse', metrics=['mae']) #Test Nadam with built in momentum
    return model
model.summary()

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
K-fold Validation
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
k = 10 #From reading the book it seems that k should be set to around 4 or 5
num_val_samples = len(X_train) // k
num_epochs = 40
all_scores = []

for i in range(k):
    print('processing fold #', i)
    val_x = X_train[i * num_val_samples: (i + 1) * num_val_samples]
    val_y = Y_train[i * num_val_samples: (i + 1) * num_val_samples]

    partial_x_train = np.concatenate (
            [X_train[:i * num_val_samples],
             X_train[(i + 1) * num_val_samples:]],
             axis=0)
    partial_y_train = np.concatenate (
            [Y_train[:i * num_val_samples],
             Y_train[(i + 1) * num_val_samples:]],
             axis=0)

    model = build_model()
    model.fit(partial_x_train, partial_y_train,
              epochs=num_epochs, batch_size = 500, verbose = 0)
    val_mse, val_mae = model.evaluate(val_x, val_y, verbose = 0)
    print("################################")
    print("MSE: ", val_mse, "MAE: ", val_mae)
```

3. Type of training (supervised, unsupervised)

Since, the objective for my model was to determine if the input variables or predictors I had were accurate at predicting diamond prices, I had created a separate Y_train contain all the price data and later a Y_test as well. Thus, the type of training I used overall was a supervised method, since my input variables had an expected output that was predetermined based off the given input attributes. You can view just a few sample rows of the diamonds dataset just showing the prices that we can view as expected outputs.



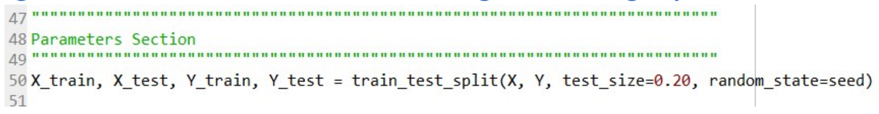
```
=============================================================================
Import Libraries Section
=============================================================================
from keras.models import Sequential
from keras.layers import Dense
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from keras import models
from keras import layers
import matplotlib.pyplot as plt
import datetime


=============================================================================
Load Data Section
=============================================================================

Diamonds = pd.read_csv('diamonds.csv', sep = ",", usecols = (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,1
DiamondsDataset = Diamonds.values

X = DiamondsDataset[:,0:23] #Setting the input variables
Y = DiamondsDataset[:,23] #Setting the output variables
start_time = datetime.datetime.now()

=============================================================================
Pretreat Data Section
=============================================================================
scaler = StandardScaler()
X = scaler.fit_transform(X)
seed = 97
np.random.seed(seed)

=============================================================================
Parameters Section
=============================================================================
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.20, random_state=seed) #Creatin
```



ble	x	y	z	price
55	3.95	3.98	2.43	326
61	3.89	3.84	2.31	326
65	4.05	4.07	2.31	327
58	4.2	4.23	2.63	334
58	4.34	4.35	2.75	335
57	3.94	3.96	2.48	336
57	3.95	3.98	2.47	336
55	4.07	4.11	2.53	337
61	3.87	3.78	2.49	337
61	4	4.05	2.39	338
55	4.25	4.28	2.73	339
56	3.93	3.9	2.46	340
61	3.88	3.84	2.33	342
54	4.35	4.37	2.71	344
62	3.79	3.75	2.27	345
58	4.38	4.42	2.68	345
54	4.31	4.34	2.68	348
54	4.23	4.29	2.7	351
56	4.23	4.26	2.71	351
59	4.21	4.27	2.66	351
56	4.26	4.3	2.71	351
55	3.85	3.92	2.48	352
57	3.94	3.96	2.41	353
62	4.39	4.43	2.62	353
62	4.44	4.47	2.59	353
58	3.97	4.01	2.41	354
57	3.97	3.94	2.47	355

4. Proportion of training and testing data sets (70:30, 80:20, etc.)

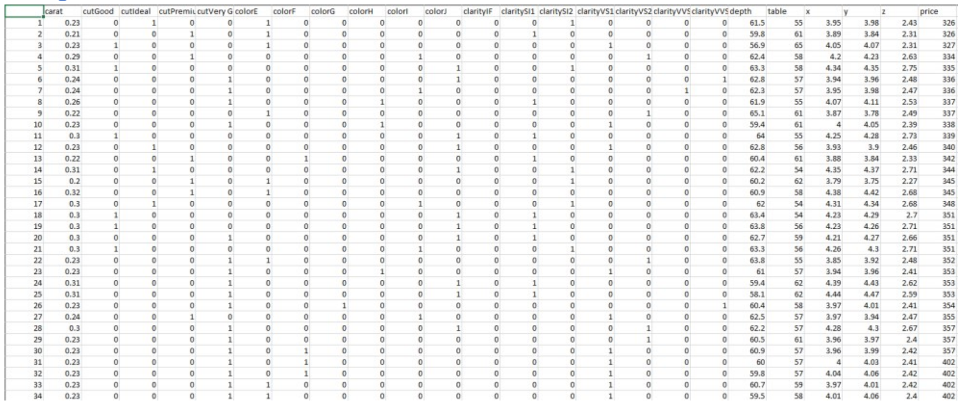
The training and testing data set that I used for my model was going off an 80:20 split. Since my dataset was so large looking back I think I honestly probably could've used a 70:30 split since I had a lot of input variables that I could test with. However, I went with a standard practice of using the 80:20 rule of thumb for training and testing my data.



```
=============================================================================
Parameters Section
=============================================================================
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.20, random_state=seed)
```

5. Number of input and output units (usually application dependent)

Due to the objective of my model being trying to determine the accuracy of my input variables being able to predict my output variable of price I wanted to use all the input variables that were provided for my dataset. My input variables consisted of carat, depth, table, x, y, z, color, clarity, cut while my output variable was the price. Since, color, clarity, and cut consisted of qualitative variables I had changed them to dummy variables to accurately utilize them as predictors. This gave me a total of 23 input variables and 1 output variable. From this image of my input and output units I do make sure to not select the first column from the dataset as we don't need the specific row IDs of each diamond.

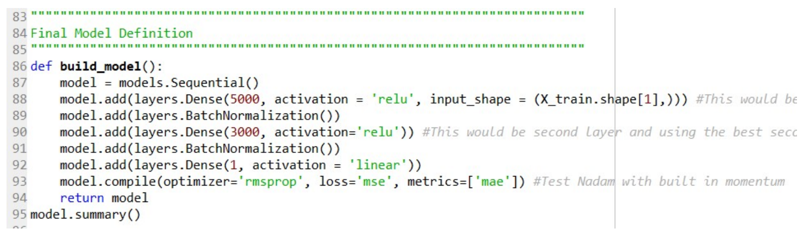


	carat	cutGood	cutIdeal	cutPremiu	cutVery G	colorE	colorF	colorG	colorH	colorI	colorJ	clarityIF	claritySI1	claritySI2	clarityVS1	clarityVS2	clarityVVS	clarityVVS	depth	table	x	y	z	price
1	0.23	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	61.5	55	3.95	3.98	2.43	326
2	0.21	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	59.8	61	3.89	3.84	2.31	326
3	0.23	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	56.9	65	4.05	4.07	2.31	327
4	0.29	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	62.4	58	4.2	4.23	2.63	334
5	0.31	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	63.3	58	4.34	4.35	2.75	335
6	0.24	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	62.8	57	3.94	3.96	2.48	336
7	0.24	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	62.3	57	3.95	3.98	2.47	336
8	0.26	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	61.9	55	4.07	4.11	2.53	337
9	0.22	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	65.1	61	3.87	3.78	2.49	337
10	0.23	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	59.4	61	4	4.05	2.39	338
11	0.3	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	64	55	4.25	4.28	2.73	339
12	0.23	0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	62.8	56	3.93	3.9	2.46	340
13	0.22	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	60.4	61	3.88	3.84	2.33	342
14	0.31	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	62.2	54	4.35	4.37	2.71	344
15	0.2	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	60.2	62	3.79	3.75	2.27	345
16	0.32	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	60.9	58	4.38	4.42	2.68	345
17	0.3	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	62	54	4.31	4.34	2.68	348
18	0.3	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	63.4	54	4.23	4.29	2.7	351
19	0.3	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	63.8	56	4.23	4.26	2.71	351
20	0.3	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	62.7	59	4.21	4.27	2.66	351
21	0.3	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	63.3	56	4.26	4.3	2.71	351
22	0.23	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	63.8	55	3.85	3.92	2.48	352
23	0.23	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	61	57	3.94	3.96	2.41	353
24	0.31	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	59.4	62	4.39	4.43	2.62	353
25	0.31	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	58.1	62	4.44	4.47	2.59	353
26	0.23	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	60.4	58	3.97	4.01	2.41	354
27	0.24	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	62.5	57	3.97	3.94	2.47	355
28	0.3	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	62.2	57	4.28	4.3	2.67	357
29	0.23	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	60.5	61	3.96	3.97	2.4	357
30	0.23	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	60.9	57	3.96	3.99	2.42	357
31	0.23	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	60	57	4	4.03	2.41	402
32	0.23	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	59.8	57	4.04	4.06	2.42	402
33	0.23	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	60.7	59	3.97	4.01	2.42	402
34	0.23	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	59.5	58	4.01	4.06	2.4	402

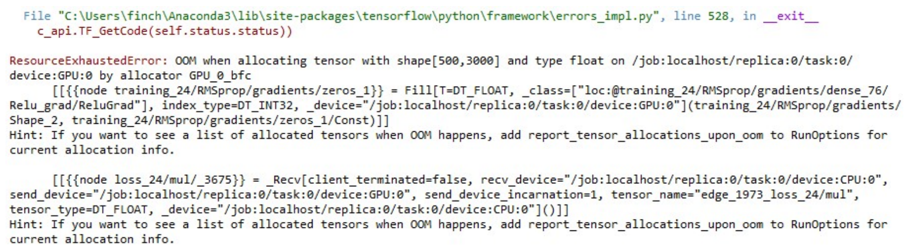
6. Number and size of hidden layers (2N+1, experimental)

While creating and improving my model I focused on an experimental approach when it came to the number and size of hidden layers I used. The number and size of hidden layers that I used for my final model was 3 hidden layers and dense layers and 2 hidden layers that was utilizing batch normalization to try and help accommodate the larger dataset that I was using and attempt to limit overfitting. I tried experimenting with additional layers and dense layers, but I did run into some errors at one point during this experimentation process and I think it was due to my input values that I was using for these layers but couldn't quite debug the error message. I

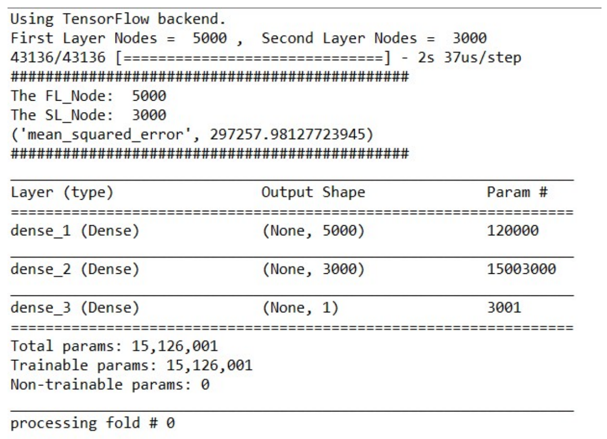
have included it below and included additional layer attempts when discussing my learning rate. I would normally only run this part multiple times when I was experimenting, so I could check my parameters each time once the model was compiled before deciding if I liked my results and wanted to move forward with the model definition.



```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Final Model Definition
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
def build_model():
    model = models.Sequential()
    model.add(layers.Dense(5000, activation = 'relu', input_shape = (X_train.shape[1],))) #This would be
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(3000, activation='relu')) #This would be second layer and using the best secc
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(1, activation = 'linear'))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae']) #Test Nadam with built in momentum
    return model
model.summary()
```



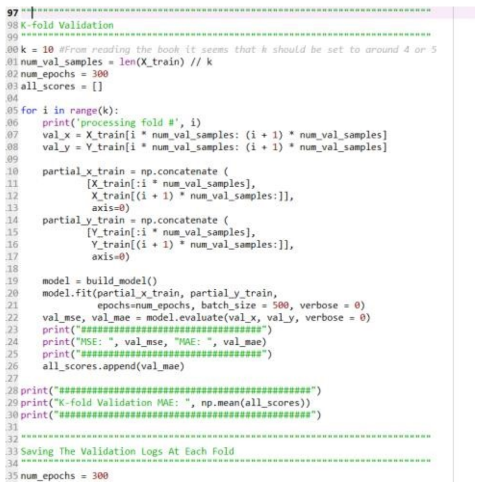
```
  File "C:\Users\finch\Anaconda3\lib\site-packages\tensorflow\python\framework\errors_impl.py", line 528, in __exit__
    c_api.TF_GetCode(self.status.status))

ResourceExhaustedError: OOM when allocating tensor with shape[500,3000] and type float on /job:localhost/replica:0/task:0/
device:GPU:0 by allocator GPU_0_bfc
	 [[{{node training_24/RMSprop/gradients/zeros_1}} = Fill[T=DT_FLOAT, _class=["loc:@training_24/RMSprop/gradients/dense_76/
Relu_grad/ReluGrad"], index_type=DT_INT32, _device="/job:localhost/replica:0/task:0/device:GPU:0"](training_24/RMSprop/gradients/
Shape_2, training_24/RMSprop/gradients/zeros_1/Const)]]
Hint: If you want to see a list of allocated tensors when OOM happens, add report_tensor_allocations_upon_oom to RunOptions for
current allocation info.

	 [[{{node loss_24/mul/_3675}} = _Recv[client_terminated=false, recv_device="/job:localhost/replica:0/task:0/device:CPU:0",
send_device="/job:localhost/replica:0/task:0/device:GPU:0", send_device_incarnation=1, tensor_name="edge_1973_loss_24/mul",
tensor_type=DT_FLOAT, _device="/job:localhost/replica:0/task:0/device:CPU:0"]()]]
Hint: If you want to see a list of allocated tensors when OOM happens, add report_tensor_allocations_upon_oom to RunOptions for
current allocation info.
```



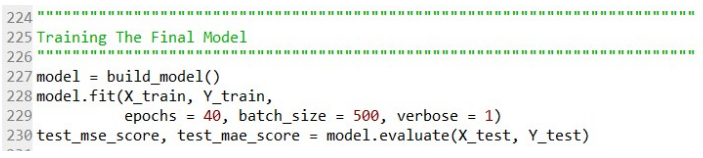
```
Using TensorFlow backend.
First Layer Nodes =  5000 ,  Second Layer Nodes =  3000
43136/43136 [==============================] - 2s 37us/step
##############################################
The FL_Node:  5000
The SL_Node:  3000
('mean_squared_error', 297257.98127723945)
##############################################
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 5000)              120000
_________________________________________________________________
dense_2 (Dense)              (None, 3000)              15003000
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 3001
=================================================================
Total params: 15,126,001
Trainable params: 15,126,001
Non-trainable params: 0
_________________________________________________________________
processing fold # 0
```

7. Number of repetitions during training (epoch)

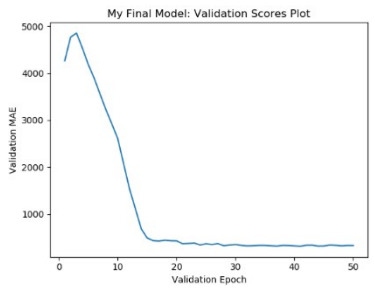
The number of epochs that I focused on for each training component of my model was set around 300 repetitions. For all my other models that I have created I used a rule of thumb and advice from class to try and train with 200 epochs. I choose 300 to see how the additional iterations would affect lowering the MSE and MAE of my model. Though as I have seen before I can see that my model begins to overfit my training data relatively early on around 50 epochs or so. Below I have included a plot when I reran my model only using 50 epochs to show this, but essentially the plot stayed the same for the 300-epoch run. However, I did like visualizing this on a graph and it was a good experimentation process to view, especially if the plot shifted at any point. For my final model since 40 epochs seemed to be the lowest MAE I achieved before overfitting this is the number of repetitions I used for training my final model.



```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
K-fold Validation
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
k = 10 #From reading the book it seems that k should be set to around 4 or 5
num_val_samples = len(X_train) // k
num_epochs = 300
all_scores = []

for i in range(k):
    print('processing fold #', i)
    val_x = X_train[i * num_val_samples: (i + 1) * num_val_samples]
    val_y = Y_train[i * num_val_samples: (i + 1) * num_val_samples]

    partial_x_train = np.concatenate (
            [X_train[:i * num_val_samples],
             X_train[(i + 1) * num_val_samples:]],
             axis=0)
    partial_y_train = np.concatenate (
            [Y_train[:i * num_val_samples],
             Y_train[(i + 1) * num_val_samples:]],
             axis=0)

    model = build_model()
    model.fit(partial_x_train, partial_y_train,
              epochs=num_epochs, batch_size = 500, verbose = 0)
    val_mse, val_mae = model.evaluate(val_x, val_y, verbose = 0)
    print("##################################")
    print("MSE: ", val_mse, "MAE: ", val_mae)
    print("##################################")
    all_scores.append(val_mae)

print("##############################################")
print("K-fold Validation MAE: ", np.mean(all_scores))
print("##############################################")

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Saving The Validation Logs At Each Fold
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
num_epochs = 300
```

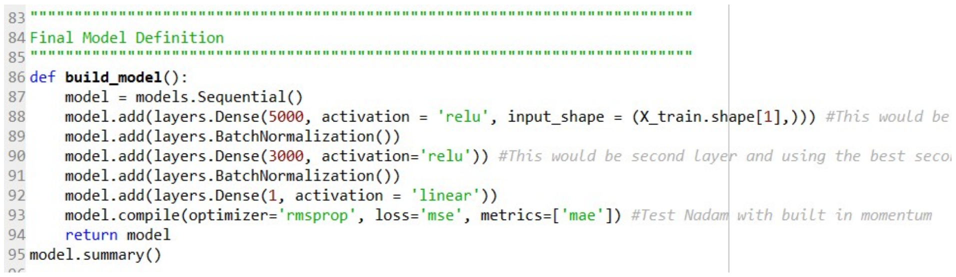


```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Training The Final Model
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
model = build_model()
model.fit(X_train, Y_train,
          epochs = 40, batch_size = 500, verbose = 1)
test_mse_score, test_mae_score = model.evaluate(X_test, Y_test)
```



8. Choice of activation function (sigmoid, linear, Tanh, relu, etc.)

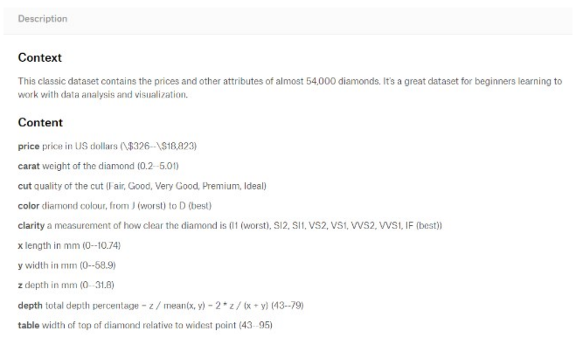
The activation choice that I used for my model was linear because I wanted to experiment while trying to predict a continuous price based on the input factors that my dataset had. Due to this I left my final layer with the linear activation function as you can see below. Since, I was looking into price I justified not using sigmoid since that would have only allowed my data to be defined and predict values that would be between 0 and 1. I also made sure to include the relu activation in my model, because I wanted to make sure that all of the negative values would be taken into consideration and flattened out, that way the continuous data that I had could be utilized more efficiently. I did not do a lot of experimentation with this process, but rather kept those decisions as constant variables when it came to try to alter and improve my model moving forward.



```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Final Model Definition
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
def build_model():
    model = models.Sequential()
    model.add(layers.Dense(5000, activation = 'relu', input_shape = (X_train.shape[1],))) #This would be
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(3000, activation='relu')) #This would be second layer and using the best seco
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(1, activation = 'linear'))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae']) #Test Nadam with built in momentum
    return model
model.summary()
```

9. Size of data set (number of records)

As you can see below the original size of my data set almost had 54,000 (53,940 exactly) lines of different diamond samples with their attributes. After I removed the additional rows that I originally had 0s in the length, width, and depth size I now had a data set size of 53,920. Again, earlier in my report I made notes on how I removed these unnecessary rows that I did not want to train my model on. Overall, I justified that it was okay to remove these 0 values because there should not be any diamond that has a 0 for these measurements and removing 20 rows out of a very large dataset should not throw off my results in a significant way.



Description

Context

This classic dataset contains the prices and other attributes of almost 54,000 diamonds. It's a great dataset for beginners learning to work with data analysis and visualization.

Content

price price in US dollars (\$326--\$18,823)

carat weight of the diamond (0.2--5.01)

cut quality of the cut (Fair, Good, Very Good, Premium, Ideal)

color diamond colour, from J (worst) to D (best)

clarity a measurement of how clear the diamond is (I1 (worst), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (best))

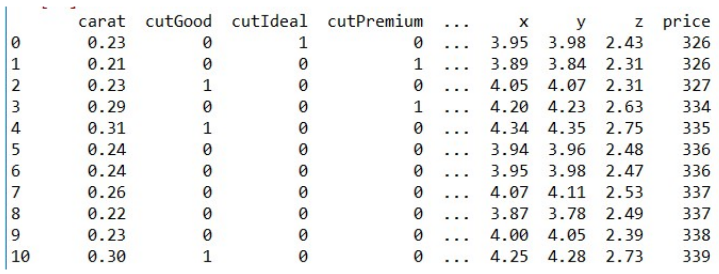
x length in mm (0--10.74)

y width in mm (0--58.9)

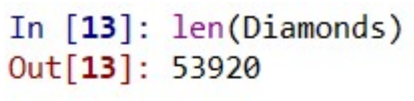
z depth in mm (0--31.8)

depth total depth percentage = z / mean(x, y) = 2 * z / (x + y) (43--79)

table width of top of diamond relative to widest point (43--95)



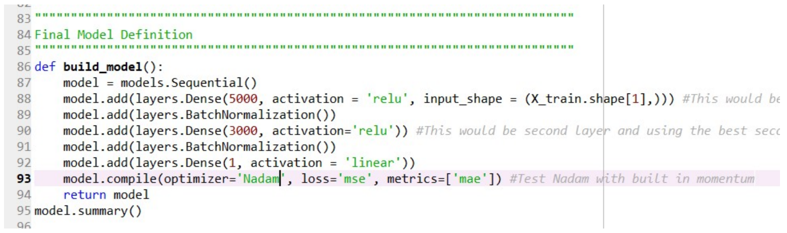
```
    carat  cutGood  cutIdeal  cutPremium  ...     x     y     z  price
0    0.23        0         1           0  ...  3.95  3.98  2.43    326
1    0.21        0         0           1  ...  3.89  3.84  2.31    326
2    0.23        1         0           0  ...  4.05  4.07  2.31    327
3    0.29        0         0           1  ...  4.20  4.23  2.63    334
4    0.31        1         0           0  ...  4.34  4.35  2.75    335
5    0.24        0         0           0  ...  3.94  3.96  2.48    336
6    0.24        0         0           0  ...  3.95  3.98  2.47    336
7    0.26        0         0           0  ...  4.07  4.11  2.53    337
8    0.22        0         0           0  ...  3.87  3.78  2.49    337
9    0.23        0         0           0  ...  4.00  4.05  2.39    338
10   0.30        1         0           0  ...  4.25  4.28  2.73    339
```



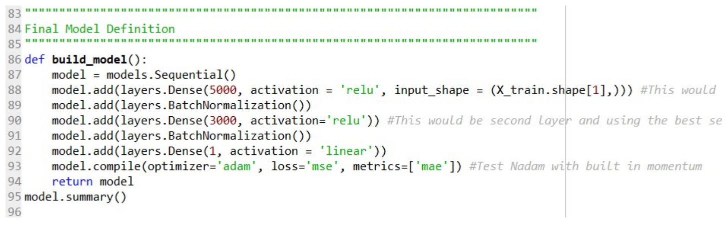
```
In [13]: len(Diamonds)
Out[13]: 53920
```

10. Learning Rate

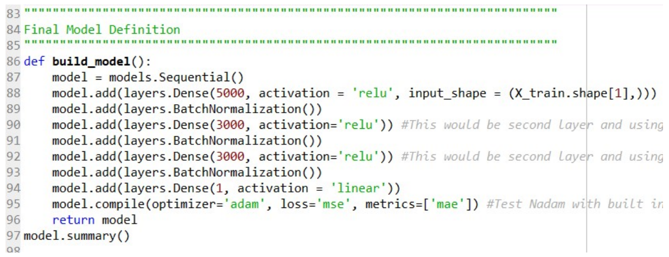
To try and improve my models learning rate I attempted several different experimentations. I wanted to attempt and use different optimizers to help the learning process, but this was also to help improve the momentum of my model as well. I got some errors working with Nadam, and I believe it is due to the type of data that my model was working with because I couldn't find a clear method to use when debugging. I also tried Adam and increased the layers and dense layers for my network to help with learning over time. I found that my model learned very well with a total of 40 epochs and a little bit after that the learning rate was just overfitting the data and the MAE wasn't really changing. I also found increasing the batch size did help improve the learning rate and essentially helped with learning quicker but didn't exactly change the MAE results too much and I stuck with a standard rate of 500 for my data.



```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Final Model Definition
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
def build_model():
    model = models.Sequential()
    model.add(layers.Dense(5000, activation = 'relu', input_shape = (X_train.shape[1],))) #This would be
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(3000, activation='relu')) #This would be second layer and using the best secc
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(1, activation = 'linear'))
    model.compile(optimizer='Nadam', loss='mse', metrics=['mae']) #Test Nadam with built in momentum
    return model
model.summary()
```



```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Final Model Definition
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
def build_model():
    model = models.Sequential()
    model.add(layers.Dense(5000, activation = 'relu', input_shape = (X_train.shape[1],))) #This would
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(3000, activation='relu')) #This would be second layer and using the best se
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(1, activation = 'linear'))
    model.compile(optimizer='adam', loss='mse', metrics=['mae']) #Test Nadam with built in momentum
    return model
model.summary()
```



```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Final Model Definition
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
def build_model():
    model = models.Sequential()
    model.add(layers.Dense(5000, activation = 'relu', input_shape = (X_train.shape[1],)))
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(3000, activation='relu')) #This would be second layer and using
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(3000, activation='relu')) #This would be second layer and using
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(1, activation = 'linear'))
    model.compile(optimizer='adam', loss='mse', metrics=['mae']) #Test Nadam with built ir
    return model
model.summary()
```

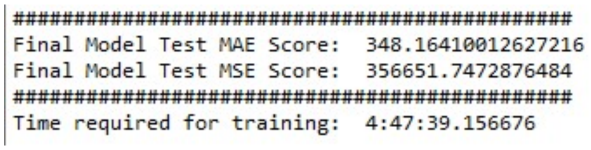
11. Momentum.

When trying to increase my model's momentum as I mentioned earlier I tried using different optimizers to help improve the momentum. I tried using Nadam too, since reading up on this from the Keras documentation website for optimizers it should act very similar to rmsprop, which I originally use however this should include increased momentum. However, I kept running into errors while utilizing the Nadam optimizer and I believe that it might be due to the type of data I am using, but I couldn't exactly figure out the error message I was getting. I would have liked to have gotten this, but overall, I was very happy with the speed that my model was training on especially for this type of neural network.

13. Discussion of results and further work.

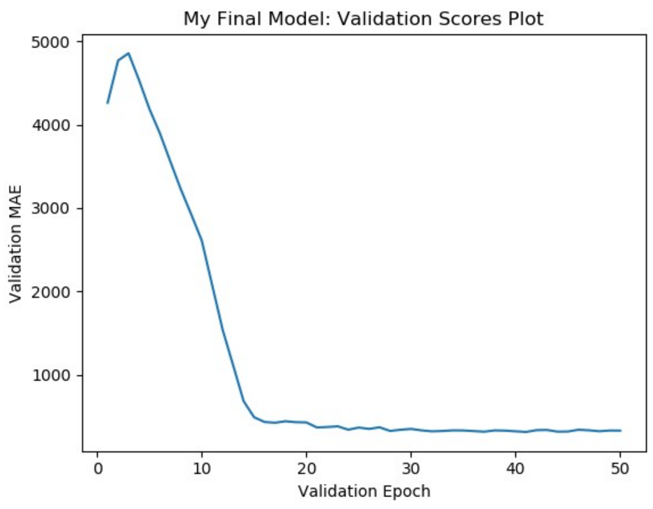
Based on the results that I was able to achieve with this regressor model I think we can reach back out to Masters Diamonds and let them know that based on their collected diamonds data set we can determine that we can make decent predictions of diamond prices based on the provided attributes. Below I have left the two different Final Model Test MAE scores from the last runs I ran for this model trained specifically on 300 epochs and then trained on 40 epochs where we determined earlier that this is the point where we begin to overfit our model. The 300epoch trained model reached a final MAE of 348.16 and the 40-epoch trained model reached a final MAE of 314.49 and although this isn't a very large change any change that helps lowers this score is more ideal. Based on all of this we can determine that the model for Masters Diamonds is still off around $314.49 when it comes to predicting the price of diamonds based on the input variables. Since, diamond prices are normally priced much higher with the average of this dataset being around $3930.99 I believe Masters Diamonds will be able to feel comfortable setting their diamond prices within a competitive range against their competitors when they officially launch this summer. The next steps that I would like to attempt in the future on this dataset is applying a multiclass classification model. This would be to potentially find the highest accuracy for the model from the input variables provided to better determine how far off we might be when adjustments are made to the inputs that are trying to predict the diamond prices.

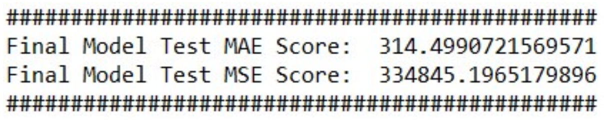
- 300 Epochs



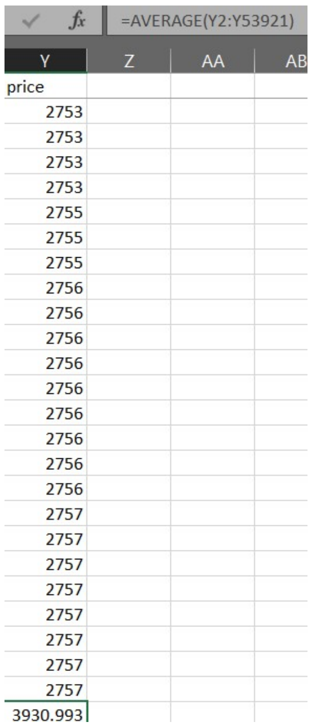
```
##################################################
Final Model Test MAE Score:  348.16410012627216
Final Model Test MSE Score:  356651.7472876484
##################################################
Time required for training:  4:47:39.156676
```

- 40 Epochs





```
##############################################
Final Model Test MAE Score:  314.4990721569571
Final Model Test MSE Score:  334845.1965179896
##############################################
```



=AVERAGE(Y2:Y53921)

Y	Z	AA	AB
price			
2753			
2753			
2753			
2753			
2755			
2755			
2755			
2756			
2756			
2756			
2756			
2756			
2756			
2756			
2756			
2756			
2757			
2757			
2757			
2757			
2757			
2757			
2757			
2757			
3930.993			

14. For each line of the code that you used for the assignment, other than those containing ‘from’ and/or ‘import’, please insert a comment above stating what each line does.
    1. Please place a comment block (code flow) below the ‘from/import’ block and above the code that describes in sentence form the overall flow of the code and the purpose of the code (what are we trying to accomplish?). You may use pseudocode, if desired.
    2. Please place a comment block below the ‘code flow’ block and above the code that lists each variable used and explains what each variable is used for.
    3. A single line comment may start with a #.
    4. A comment block should start with ‘’’ and end with ‘’’.
    5. If you are commenting two or more lines of code that are essentially identical, i.e. different variables but identical operations, you may use one comment above that code block. For code that has similar operations (model. add), but has different parameters, please comment each line.
    6. Please do not combine comment blocks.

Below is my code that I have added comments to during each step. Since I mainly used the same script but modified it each time I only left the results that I had to save space on this document. Though if I made changes on certain lines, I made sure to include that in the comment what it would have looked like if I kept the copy during the experimentation steps.

My Code:

```
        # -*- coding: utf-8 -*- """
        Created on Sun Apr 7 13:43:50 2019

        @author: Masters
        """
        #Used this to research some keras operations especially for optimizers and
momentum
        #https://keras.io/optimizers/
        """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""""""
        Final Project
        """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""""""

        """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""""""
        Import Libraries Section
        """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""" from keras.models import
        Sequential from keras.layers import
        Dense import numpy as np import
        pandas as pd
        from sklearn.preprocessing import StandardScaler
        from sklearn.model_selection import train_test_split
        from keras import models from keras import layers
        import matplotlib.pyplot as plt
        import datetime
```

""
"""""""""""""""

Load Data Section

""
"""""""""""""""

#I am using pandas to read in my diamonds csv file from kaggle and calling the specific columns I want to use. This is where I make sure not to bring in the first column, which just contained the specific row IDs.

Diamonds = pd.read_csv('diamonds.csv', sep = ",", usecols = (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24), header=0)

#I am saving the specific values of the csv file into an array of values

DiamondsDataset = Diamonds.values

#From the list that I created with all the values I am setting aside the input variables into a X variable

X = DiamondsDataset[:,0:23] #Setting the input variables

#From the list that I created with all the values I am setting aside the output variables into a Y variable

Y = DiamondsDataset[:,23] #Setting the output variables

#Using the datetime import, I am using this to start the timer for how long it takes my model to run start_time = datetime.datetime.now()

""
"""""""""""""""

Pretreat Data Section

""
"""""""""""""""

#Before being able to use my input variables or X at this point I need to pretreat my data and to do this I need to scale the data. Setting the StandardScaler to a scaler variable to call next.

scaler = StandardScaler()

#Using the fit_transform function from scaler I can now transform or scale my X (input variables) to use for my model.

X = scaler.fit_transform(X)

#Creating a seed variable to use for setting a random seed seed

= 97

#Setting the random seed to 97 (My favorite Ice Hockey number!)

np.random.seed(seed)

""
"""""""""""""""

Parameters Section

""
"""""""""""""""

#Creating the train and test sets for my model and focusing on a 80/20 split for the test data.

```
        X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.20,
random_state=seed)

        """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""
        Experiment Model To Find Best Nodes To Use
        """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""
        #Creating a variable for my first layer node #I changed the weights to experiment in
minimizing the loss function #First tried [10, 20, 30, 40, 50, 60, 70, 80, 90, 100] then
increased it much higher before just leaving the best value that I found.
        FL_Nodes=[5000]
        #Creating a variable for my second layer node #I changed the weights to experiment
in minimizing the loss function #First tried [10, 20, 30, 40, 50] and then increased it much
higher before just leaving the best value that I found.
        SL_Nodes=[3000]
        #I am using this for loop to iterate through the list of First Layer nodes that
I created above during my experimentations for Num_Nodes_FL in FL_Nodes:
          #I am using this for loop to iterate through the list of Second Layer nodes that
I
        created above during my experimentations     for Num_Nodes_SL in SL_Nodes:
            #Printing the First Layer and Second Layer Nodes selected #This was mainly to
help me during the experimentation process, so I knew exactly what node selections were
getting the specific results.
            print("First Layer Nodes = ", Num_Nodes_FL, ", ", "Second Layer Nodes = ",
Num_Nodes_SL)
            #Since we are starting the same model each time I am using this function to
help create a new model for each iteration.
            model = Sequential()
            #Adding a layer and dense layer that is based on the first layer selection from
the for loop and setting the input dim to the length of the input variables.
            model.add(Dense(Num_Nodes_FL, input_dim = 23, activation = 'relu'))
            #Checking to make sure that the iteration in the Second Layer nodes is greater
than 0 and if so we will continue.
            if Num_Nodes_SL > 0:
              #Adding a layer and dense layer that is based on the second layer selection
from the for loop.
              model.add(Dense(Num_Nodes_SL, activation = 'relu'))
            #Adding a final layer and setting the activation to linear for this regressor
model.
            model.add(Dense(1, activation = 'linear'))


"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""

            #Train Model Section
```

""
"""""""

	#Now that I have my model created from earlier I am working on compiling the model here to train it. #I am focusing on MSE as the metrics and loss while using the rmsprop as the optimizer, which is common for this type of regressor model.
	model.compile(loss = 'mse', optimizer = 'rmsprop', metrics = ['mse'])
	#At this stage I am fitting the experimentation model with the train data and using 300 epochs and a batch size of 500. I explain my choice on these values more in my report.
	model.fit (X_train, Y_train, epochs = 300, batch_size = 500, verbose = 0)

""
"""""""

	#Show Output Section

""
"""""""

	#Now that I have trained my model I am evaulating the results based on the X and Y train and saving the scores into a score variable.
	scores = model.evaluate(X_train, Y_train)
	print("##")
	#I wanted to print the first layer node that we are iterating through, so I know exactly what selection gets the score that is printed below.
	print("The FL_Node: ", Num_Nodes_FL)
	#I wanted to print the second layer node that we are iterating through, so I know exactly what selection gets the score that is printed below.
	print("The SL_Node: ", Num_Nodes_SL)
	#Finally, I am printing the score that belongs with the First and Second Layer nodes, so I can compare the best results to use for my official model print((model.metrics_names[1], scores[1]))
	print("##")
	#This concludes my experimentation process to find the best layer units that I plan to use for my Final model

""
"""""""""""""""

Final Model Definition
""
"""""""""""""""

#At this step I am building my final model based on the experimentation results that I found from above. On my report I include the different attempts and results I got, but I just left my final selections here.
def build_model():
  model = models.Sequential()
  #Here I am adding the best layer unit from my experimentation process I got earlier for my first layer   model.add(layers.Dense(5000, activation = 'relu', input_shape =
(X_train.shape[1],)))

#From the book I looked up batch normalization, which seems to help prevent overfitting with larger datasets, which I wanted to try and use when building my final model here.
model.add(layers.BatchNormalization())
#Here I am adding the best layer unit from my experimentation process I got earlier for my second layer model.add(layers.Dense(3000, activation='relu'))
#From the book I looked up batch normalization, which seems to help prevent overfitting with larger datasets, which I wanted to try and use when building my final model here.
model.add(layers.BatchNormalization())
#Adding a final layer and setting the activation to linear for this regressor model for continious learning model.add(layers.Dense(1, activation = 'linear'))
#Finally, I am compiling everything similar during my experimentation process #I also attempted to use Nadam and adam as different optimizers when experimenting with the learning rate and momentum.
model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
return model
#Now that everything has been created for my final model I am checking the summary of my model that I have put together.
model.summary()

""
K-fold Validation
""
#In order to validate my model and data I wanted to use K-Fold validation for this process. Based on the different runs and different validations we can find out if we are overfitting our data and the best number of epochs that we could use for training our final model.
#I am creating a variable k to store the number of k-folds to go through. From reading up on this in the book it seems the recommended Ks to use is 4 to 5, but I wanted to attempt 10 to see the results.
k = 10 #From reading the book it seems that k should be set to around 4 or 5
#Creating a num_val_samples that is the length of the X_train data divided by the total number of Ks I am using. This will be later used to create the validation data.
num_val_samples = len(X_train) // k
#Creating a number of epochs variable to use for training num_epochs = 300
#Creating an empty list to store all the scores that each iteration of K is creating during this validation all_scores = []

#Using a for loop to go through the number of Ks that I selected for i in range(k):
#Just printing the iteration of k that is being processed, so I can view with results
print('processing fold #', i)
#Creating a validation data set based on the X_train data and on the partition of K that is chosen along with the number of validation samples we need.

```
            val_x = X_train[i * num_val_samples: (i + 1) * num_val_samples]
            #Creating a validation data set based on the Y_train data and on the partition of
K that is chosen along with the number of validation samples we need.
            val_y = Y_train[i * num_val_samples: (i + 1) * num_val_samples]

            #Creating a partial x train variable that is based on the training data from the
other partitions that are outside the val_x and val_y variables we created above.
            partial_x_train = np.concatenate ([X_train[:i * num_val_samples], X_train[(i + 1)
* num_val_samples:]], axis=0)
            #Creating a partial y train variable that is based on the training data from the
other partitions that are outside the val_x and val_y variables we created above.
            partial_y_train = np.concatenate ([Y_train[:i * num_val_samples], Y_train[(i + 1)
* num_val_samples:]], axis=0)

            #Now that all the validation data and training is created I can build my model.
         model = build_model()
            #I am fitting my model for each K iteration based on the partial x and partial y
data     model.fit(partial_x_train, partial_y_train, epochs=num_epochs, batch_size = 500,
verbose = 0)
            #Based on the fit of the model I am now evaulting the validation x and validation
y data and saving to two scores MSE & MAE to view.
            val_mse, val_mae = model.evaluate(val_x, val_y, verbose = 0)
            print("###################################")
            #This is where I am viewing the specific MSE and MAE scores that I get for each
K iteration to help determine the best results
         print("MSE: ", val_mse, "MAE: ", val_mae)
         print("###################################")
            #Since we still have the empty list that was created before I am saving all the
validation MAE results into this list to find the mean of all of them next.
            all_scores.append(val_mae)

         #Now I can view the average validation MAE from this K-fold validation process
         print("################################################") print("K-fold
         Validation MAE: ", np.mean(all_scores))
         print("################################################")

         """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""""""""

         Saving The Validation Logs At Each Fold
         """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""""""""

         #After performing the K-fold validation I wanted to now keep track of how well the
above model does at each epoch to help with determining the number of epochs to use
when training the final model num_epochs = 300
         #This time creating an empty list for all the mae results each time
         all_mae_histories = []
         #Using a for loop to go through the number of Ks that I selected  for
         i in range(k):
```

```
        #Just printing the iteration of k that is being processed, so I can view with results
      print('processing fold #', i)

        #Creating a validation data set based on the X_train data and on the partition of
K that is chosen along with the number of validation samples we need.
        val_x = X_train[i * num_val_samples: (i + 1) * num_val_samples]
        #Creating a validation data set based on the Y_train data and on the partition of
K that is chosen along with the number of validation samples we need.
        val_y = Y_train[i * num_val_samples: (i + 1) * num_val_samples]

        #Creating a partial x train variable that is based on the training data from the
other partitions that are outside the val_x and val_y variables we created above.
        partial_x_train = np.concatenate ([X_train[:i * num_val_samples], X_train[(i + 1)
* num_val_samples:]], axis=0)
        #Creating a partial y train variable that is based on the training data from the
other partitions that are outside the val_x and val_y variables we created above.
        partial_y_train = np.concatenate ([Y_train[:i * num_val_samples], Y_train[(i + 1)
* num_val_samples:]], axis=0)

        #Now that all the validation data and training is created I can build my model.
        model = build_model()

        #This time however I am training the model using the partial x and y variable and
incorporating the validation data to fit as well and saving everything into a history variable
to keep track of the log.
        history = model.fit(partial_x_train, partial_y_train, validation_data = (val_x,
val_y), epochs=num_epochs, batch_size=500, verbose = 0)
        #From the history variable that was created above I can now pull out the
validation MAE that has been recorded each time.
        mae_history = history.history['val_mean_absolute_error']
        #Since we still have the empty list that was created before I am saving all the
mae_history results into this list.
        all_mae_histories.append(mae_history)

      """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""""""""
      Building The History of Successive Mean K-Fold Validation Scores
      """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""""""""
      #Now I am able to calculate the average of each epoch MAE score that was found
for each of the iterations beforehand.
      average_mae_history = [np.mean([x[i] for x in all_mae_histories]) for i in
range(num_epochs)]
      #Once this is computed I can save the mean of the entire MAE history and print it
to view during this step.
      print("The average mae history is: ", average_mae_history)

      """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
```

""""""""""""""""

Plotting Validation Scores

""
""""""""""""""""

#Since I have the average MAE history results each time I can now plot the validation scores during the training and experimentation. This allows me to view the average MAE for each epoch over time and can see the point where overfitting could potentially happen.

plt.plot(range(1, len(average_mae_history) + 1), average_mae_history)
#Here I am adding a title for my plot
plt.title('My Final Model: Validation Scores Plot')
#Here I am adding a x label to my plot
plt.xlabel('Validation Epoch') #Here I
am adding a y label to my plot
plt.ylabel('Validation MAE')
#I am using plot figure, since I am trying to plot additional charts and want to view them all at once.
plt.figure()

""
""""""""""""""""

#Creating A Plot To Show The Training And Validation Loss

""
""""""""""""""""

#Since I created a history variable to save all the results from fitting the partial x and y data against the validation data I can plot the training vs validation loss.

#Creating a loss variable based on the loss of history
loss = history.history['loss']
#Creating a val_loss variable based on the validation loss of history val_loss
= history.history['val_loss']
#Setting the epochs to the length of loss and adding 1 to keep track of the epochs for plot epochs = range(1, len(loss) + 1) #Plotting the training loss
plt.plot(epochs, loss, 'bo', label = 'Training Loss')
#Plotting the validation loss
plt.plot(epochs, val_loss, 'b', label = 'Validation Loss')
#Setting a title for the plot
plt.title('Plotting the Training and Validation Loss')
#Setting a x label for the plot plt.xlabel('Epochs')
#Setting a y label for the plot plt.ylabel('Loss')
#Setting a legend for the plot plt.legend()
#I am using plot figure, since I am trying to plot additional charts and want to view them all at once plt.show()

""
""""""""""""""""

Training The Final Model

""
""""""""""""""""

#Once I found my results from the plots and experimentation above I can now train my final model to get the lowest MAE that I can find.

#Finally, I am creating one last model to train my data on model
= build_model()

#I am fitting my model on the X and Y train data and setting the number of epochs to the best I found that seemed to be where overfitting began, which was around 40 model.fit(X_train, Y_train, epochs = 40, batch_size = 500, verbose = 1)

#Once my model is done fitting the training data I can evaulate the data on the X and Y test data test_mse_score, test_mae_score = model.evaluate(X_test, Y_test)

**"""
"""""""""**

Final Test MAE Score
**"""
"""""""""**

#With the final model created, fitted, and evaulated I am printing the test MAE & MSE scores that I achieved from this below
print("##")
print('Final Model Test MAE Score: ', test_mae_score) print('Final Model Test MSE Score: ', test_mse_score)
print("##")

**"""
"""""""""**

My Model Time Section
**"""
"""""""""**

#This is to stop the timer that I started at the beginning of my model stop_time
= datetime.datetime.now()

#I am printing the time that it took to train my entire model, which helped with the learning rate & momentum aspect too of trying to speed up everything.

print ("Time required for training: ",stop_time - start_time)