Wine Quality ANN Regressor

By Koby Arndt

**Objective**

The objective of this ANN is to predict the value of the quality variable based on the values of all other variables in the dataset. Quality is an ordinal variable which we will treat as continuous, so this is a regression instead of a classification problem.

**Final ANN Model in Code**

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

Define Model Section

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

def build\_model():

model = Sequential()

model.add(Dense(layer1\_nodes, activation= act\_function, input\_shape=(X.shape[1],)))

model.add(Dense(layer2\_nodes, activation= act\_function))

model.add(Dense(layer2\_nodes, activation= act\_function))

model.add(Dense(1))

model.compile(loss='mean\_squared\_error', optimizer=optimizer, metrics=['mae'])

return model

**Final Model and Training Algorithm in Words**

The final ANN model consists of one input layer, two hidden layers with, and an output layer. The input and hidden layers use the relu activation function and have 150 nodes each. The final model uses the adam optimizer. The number of epochs, i.e. the number of times the entire dataset is passed through the model per run, was set to 150. The batch size, the number of training examples included in each forward pass, was also set to 300.

**Experimental Plan**

There were many potential tuning parameters to experiment with in this model. The analyst tested different numbers of hidden layers and nodes, activation functions, and optimizers before settling on the final model. Each model attribute was considered one at a time, although given more time, a more complex method of model selection could have been used. For all candidate models for each parameter, a table of their average validation MSE, test MSE, and runtime will be given. Higher values for all metrics will be colored red and lower values, being more desirable, will be colored blue.

The initial model included one hidden layer with 15 nodes. This model achieved an average validation MSE of 0.01472 and a test MSE of 0.01492 with a runtime of 1.717 seconds.

**Nodes and Layers**

It was first considered whether a greater number of layers and nodes would increase the predictive power of the model. A model with two hidden layers and 150 nodes for each non-output layer was tested. This model achieved an average validation MSE of 0.01344 and a test MSE of 0.0131 at a runtime of 5.147 seconds.

Retaining the hidden layers but reducing the number of nodes to 15 for each resulted in a model achieving an average validation MSE of 0.0142 and a test MSE of 0.0146 with a runtime of 3.212 seconds.

Increasing the number of nodes to 300 resulted in an average validation MSE of 0.01335 and a test MSE of 0.01335 with a runtime of 8.837 seconds.

Increasing the number of nodes to 500 resulted in an average validation MSE of 0.01329 and a test MSE 0.01359 with a runtime of 23.120 seconds. At this point it was clear that increasing the number of nodes was increasing runtime without significantly impacting error metrics. The 150-node, four-layer network was chosen for its tradeoff in improvement on error metrics and slightly longer runtime.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Metrics** | **3-layer 15 nodes** | **4-layer 150 nodes** | **4-layer 15 nodes** | **4-layer 300 nodes** | **4-layer 500 nodes** |
| Mean Val MSE | 0.01472 | 0.01344 | 0.0142 | 0.01335 | 0.01329 |
| Test MSE | 0.01492 | 0.0131 | 0.0146 | 0.01335 | 0.01359 |
| Runtime (s) | 1.717 | 5.147 | 3.212 | 8.837 | 23.12 |

**Figure 1**: Comparison of candidate models tuning nodes and layers

**Number of Epochs**

The second tuning parameter to be considered was the number of epochs. Using the same four-layer network, this time at 150 epochs, the model achieved an average validation MSE of 0.01421 and a test MSE of 0.01494 with a runtime of 14.531 seconds.

Using the same network, at 300 epochs, the model achieved an average validation MSE of 0.01439 and a test MSE of 0.0155 with a runtime of 33.1059 seconds.

Using the same network, at 100 epochs, the model achieved an average validation MSE of 0.01337 and a test MSE of 0.01441 with a runtime of 9.783 seconds. Higher numbers of epochs resulted in iteratively decreasing MSEs, but without a decrease in k-fold validation MSE. This indicates that further reduction in MSE was a result of overfitting the training set. Therefore, the number of epochs was restrained to avoid overfitting.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Metrics** | **50 epochs** | **150 epochs** | **300 epochs** | **100 epochs** |
| Mean Val MSE | 0.01344 | 0.01421 | 0.01439 | 0.01337 |
| Test MSE | 0.0131 | 0.01494 | 0.0155 | 0.01441 |
| Runtime (s) | 5.147 | 14.531 | 33.1059 | 9.783 |

**Figure 2**: Comparison of candidate models tuning number of epochs

**Batch Size**

The third tuning parameter to be considered is batch size. We will not consider a batch size lower than 50 in the interest of speed. Increasing the batch size to 300 resulted in the model achieving an average validation MSE of 0.0134 and a test MSE of 0.0131 with a runtime of 4.029 seconds.

Increasing the batch size to 500 resulted in the model achieving an average validation MSE of 0.01321 and a test MSE of 0.01361 with a runtime of 3.084 seconds. Although this is an improvement in both validation MSE and runtime, the improvements are marginal at this point, so we will prefer the 300-batch model with its lower test MSE.

|  |  |  |  |
| --- | --- | --- | --- |
| **Metrics** | **Batch 50** | **Batch 300** | **Batch 500** |
| Mean Val MSE | 0.01337 | 0.0134 | 0.01321 |
| Test MSE | 0.01441 | 0.0131 | 0.01361 |
| Runtime (s) | 9.783 | 4.029 | 3.084 |

**Figure 3**: Comparison of candidate models tuning batch size

**Activation Function**

The fourth tuning parameter to consider is the activation function. Using the elu Exponential Linear Unit activation function, the model achieved an average validation MSE of 0.0150 and a test MSE of 0.0137 with a runtime of 3.331 seconds.

Using the selu activation function, the model achieved an average validation MSE of 0.01387 and a test MSE of 0.01496 with a runtime of 3.355 seconds. Both candidate activation functions performed worse on validation and test MSE, so we will prefer the relu activation function.

|  |  |  |  |
| --- | --- | --- | --- |
| **Metrics** | **relu** | **elu** | **selu** |
| Mean Val MSE | 0.0134 | 0.015 | 0.01387 |
| Test MSE | 0.0131 | 0.0137 | 0.01496 |
| Runtime (s) | 4.029 | 3.31 | 3.355 |

**Figure 4**: Comparison of candidate models tuning activation function

**Optimizer**

The fifth and final tuning parameter to consider is the optimizer. Using the RMSprop optimizer, the model achieved an average validation MSE of 0.01494 and a test MSE of 0.01434 with a runtime of 4.230 seconds. This is a disimprovement in both validation and test MSE, so we will prefer the adam optimizer.

|  |  |  |
| --- | --- | --- |
| **Metrics** | **adam** | **RMSprop** |
| Mean Val MSE | 0.0134 | 0.01494 |
| Test MSE | 0.0131 | 0.01434 |
| Runtime (s) | 4.029 | 4.23 |

**Figure 5**: Comparison of candidate models tuning optimizer

**Input Variables and Preprocessing**

Parameters were included as input variables to simplify tuning in the model selection process. The number of epochs, columns, nodes, and K-folds are all included as input variables. The original data file name, activation function, optimizer, and batch size are also included.

**Input variables in code:**

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

Parameters Section

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

num\_epochs = 100

csv\_file = 'winequality-white.csv'

num\_cols = 11

num\_cols\_scale = 11

layer1\_nodes = 150

layer2\_nodes = 150

act\_function = 'relu'

optimizer = 'adam'

set\_verbose = 2

num\_folds = 10

batch\_size = 300

All variables in the dataset were preprocessed before being used for model fitting. This is because there are a variety of variables in the dataset of various scales and distributions. Preprocessing places all variables on the same scale so that relative rather than absolute differences can be learned by the neural network. The data was then split into a training set consisting of 80% of the data and a testing set consisting of the remaining 20%.

**Preprocessing in code:**

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

Pretreat Data Section

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

dataframe = dataframe.replace(np.nan, 0)

dataset = dataframe.values

X = dataset[:,0:num\_cols]

Y = dataset[:,num\_cols]

X\_MinMax = preprocessing.MinMaxScaler()

Y\_MinMax = preprocessing.MinMaxScaler()

Y=np.array(Y).reshape(len(Y),1)

X = X\_MinMax.fit\_transform(X)

Y = Y\_MinMax.fit\_transform(Y)

print(X\_MinMax.scale\_)

print(Y\_MinMax.scale\_)

print(X.shape)

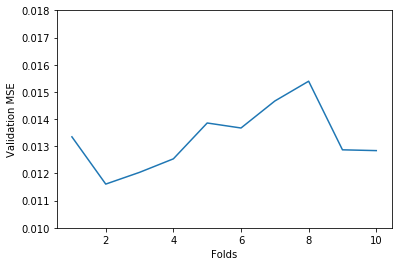
X, X\_test, Y, Y\_test = train\_test\_split(X, Y, test\_size=0.2)

**Error Metrics and Justification**

Mean squared error was used as the loss function, the objective that the neural network seeks to minimize. Mean absolute error was also considered as a metric. These are both standard regression error metrics that, through squaring or absolute value, consider the distance between the prediction and the true value regardless of positive or negative directionality.

**Model Validation**

The model was validated through a K-fold cross validation. K was set to ten. This means that the data is split into ten groups, and each group is iteratively used as a testing set while the other nine serve as the training set. K-fold validation was critical in judging the exact nature of improvements in the loss function brought about by model experimentation. Almost regardless of parameters tuned, validation MSE tended to fluctuate between 0.011 and 0.017. Higher ranges were given by particularly poor models, but no model ever improved beyond this range. This gives an idea of the model’s irreducible error--experimental models which decreased training MSE past this point were generally overfitting.



**Figure 6**: Plot of validation MSE of the final model over 10 folds

**K-fold validation in code:**

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

K-fold validation

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

X\_val = np.concatenate((X, X\_test), axis = 0)

Y\_val = np.concatenate((Y, Y\_test), axis = 0)

acc\_per\_fold = []

loss\_per\_fold = []

# Define K-Fold

kfold = KFold(n\_splits = num\_folds, shuffle = True)

fold\_no = 1

for train, test in kfold.split(X\_val,Y\_val):

model = build\_model()

print(f'Training for fold {fold\_no} ...')

history = model.fit(X\_val[train], Y\_val[train], batch\_size=batch\_size,

epochs = num\_epochs, verbose = set\_verbose)

scores = model.evaluate(X\_val[test], Y\_val[test], verbose = 0)

print(f'Fold {fold\_no} score: {model.metrics\_names[0]} of {scores[0]}; {model.metrics\_names[1]} of {scores[1]\*100}')

acc\_per\_fold.append(scores[1]\*100)

loss\_per\_fold.append(scores[0])

fold\_no = fold\_no + 1

**Results**

The final model achieved a test MAE of 0.08931. The Mean Absolute Error (MAE) indicates the average difference between the predicted quality and the true quality. Therefore, on average, the final model predicted wine quality with over 91% accuracy.

**Appendix**

**Item A: Full model in code**

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

Import Libraries Section

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

import datetime

import numpy as np

import pandas as pd

from tensorflow import keras

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from matplotlib import pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn import preprocessing

from sklearn.model\_selection import KFold

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

Parameters Section

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

num\_epochs = 100

csv\_file = 'winequality-white.csv'

num\_cols = 11

num\_cols\_scale = 11

layer1\_nodes = 150

layer2\_nodes = 150

act\_function = 'relu'

optimizer = 'adam'

set\_verbose = 2

num\_folds = 10

batch\_size = 300

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

Load Data Section

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

dataframe = pd.read\_csv(csv\_file, delimiter = ";", header = None)

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

Pretreat Data Section

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

dataframe = dataframe.replace(np.nan, 0)

dataset = dataframe.values

X = dataset[:,0:num\_cols]

Y = dataset[:,num\_cols]

X\_MinMax = preprocessing.MinMaxScaler()

Y\_MinMax = preprocessing.MinMaxScaler()

Y=np.array(Y).reshape(len(Y),1)

X = X\_MinMax.fit\_transform(X)

Y = Y\_MinMax.fit\_transform(Y)

print(X\_MinMax.scale\_)

print(Y\_MinMax.scale\_)

print(X.shape)

X, X\_test, Y, Y\_test = train\_test\_split(X, Y, test\_size=0.2)

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

Define Model Section

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

def build\_model():

model = Sequential()

model.add(Dense(layer1\_nodes, activation= act\_function, input\_shape=(X.shape[1],)))

model.add(Dense(layer2\_nodes, activation= act\_function))

model.add(Dense(layer2\_nodes, activation= act\_function))

model.add(Dense(1))

model.compile(loss='mean\_squared\_error', optimizer=optimizer, metrics=['mae'])

return model

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

K-fold validation

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

X\_val = np.concatenate((X, X\_test), axis = 0)

Y\_val = np.concatenate((Y, Y\_test), axis = 0)

acc\_per\_fold = []

loss\_per\_fold = []

# Define K-Fold

kfold = KFold(n\_splits = num\_folds, shuffle = True)

fold\_no = 1

for train, test in kfold.split(X\_val,Y\_val):

model = build\_model()

print(f'Training for fold {fold\_no} ...')

history = model.fit(X\_val[train], Y\_val[train], batch\_size=batch\_size,

epochs = num\_epochs, verbose = set\_verbose)

scores = model.evaluate(X\_val[test], Y\_val[test], verbose = 0)

print(f'Fold {fold\_no} score: {model.metrics\_names[0]} of {scores[0]}; {model.metrics\_names[1]} of {scores[1]\*100}')

acc\_per\_fold.append(scores[1]\*100)

loss\_per\_fold.append(scores[0])

fold\_no = fold\_no + 1

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

Show & Plot output Section

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

print(np.mean(loss\_per\_fold))

print(min(loss\_per\_fold))

print('Minimum loss at fold:', loss\_per\_fold.index(min(loss\_per\_fold)))

plt.plot(range(1, len(loss\_per\_fold) + 1), loss\_per\_fold)

plt.xlabel('Folds')

plt.ylabel('Validation MSE')

axes = plt.gca()

axes.set\_ylim([0.0100,0.0180])

plt.show()

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

Final Model Application

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

start\_time = datetime.datetime.now()

model = build\_model()

test\_mse\_score = []

estimator = model.fit(X, Y, epochs=num\_epochs, batch\_size=batch\_size, verbose=set\_verbose)

test\_mse\_score, test\_mae\_score = model.evaluate(X\_test, Y\_test)

print('test MSE: ',test\_mse\_score)

print('Mean Val MSE: ',np.mean(loss\_per\_fold))

stop\_time = datetime.datetime.now()

print("Time:", stop\_time - start\_time)

plt.plot(estimator.history['loss'])

plt.plot(estimator.history['mae'])

plt.show()