chapter-3-3

May 2, 2022

1 NN Heston Model - Model Training

```
In [63]: import pandas as pd
         from sklearn.model_selection import train_test_split
```

1.1 Data Preprocessing

Initialization

```
In [64]: import re

         verbose    = True
         TAG        = '0000'

         inFile     = "full_%s.csv" %(TAG)
         scalerFile = "scaler_%s.pkl" %(TAG)
         mdlDir     = "model_%s.krs" %(TAG)

         resFile    = re.sub("\..*$","_trained.png", inFile)
         print("%s -> %s" %(inFile, resFile))
```

```
full_0000.csv -> full_0000_trained.png
```

Read the training DB

```
In [65]: # Read in training data
         print("@ %-24s: reading from '%s'" %("Info", inFile))
         db = pd.read_csv(inFile, sep=',')
```

```
@ Info                    : reading from 'full_0000.csv'
```

check that it is is what we expect

```
In [66]: print("*"*82+"\n"+"* X"); print(db.keys()); print("*"*82)
         print(db.head(4))
```

```
********************************************************************************
* X
Index(['k', 'theta', 'sigma', 'v0', 'rho', 'T', 'Strike', 'Price'], dtype='object')
********************************************************************************
          k     theta     sigma        v0       rho         T    Strike  \
0  0.509975  0.130755  0.139814  0.778713 -0.505420  0.687304  1.077324
1  0.571375  0.446076  0.937467  0.768080 -0.720002  0.672622  0.855332
2  0.279107  0.493689  0.604936  0.331763 -0.898390  0.401510  1.128948
3  0.190888  0.687982  0.438101  0.781273 -0.393282  1.017852  1.381108

      Price
0  0.316269
1  0.179939
2  0.224901
3  0.608431
```

```
In [67]: from sklearn.preprocessing import StandardScaler

         '''
             It is critical that any data preparation performed on a training dataset is also pe
             on a new dataset in the future. This may include a test dataset when evaluating a m
             or new data from the domain when using a model to make predictions. Typically, the
             on the training dataset is saved for later use. The correct solution to preparing n
             for the model in the future is to also save any data preparation objects, like data
             to file along with the model.
         '''

         def preprocess(**keywrds):

             db = keywrds["db"]


             # Specify the target labels and flatten the array
             #t=np.ravel(db["Price"])
             t=db["Price"]

             # Specify the data
             X = db.drop(columns="Price")

             print("Info")
             print(X.info())

             print("Head")
             print(X.head(n=2))
             print("Tail")
             print(X.tail(n=2))
```

```python
    print("Describe")
    print(X.describe())


    # Define the scaler
    scaler = StandardScaler().fit(X)

    # Split the data up in train and test sets
    X_train, X_test, t_train, t_test = train_test_split(X, t, test_size=0.33, random_st

    # Scale the train set
    X_train = scaler.transform(X_train)

    # Scale the test set
    X_test = scaler.transform(X_test)

    return scaler, X_train, X_test, t_train, t_test
```

In [68]:
```python
scaler, X_train, X_test, t_train, t_test = preprocess(db = db)
```

```
Info
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 66978 entries, 0 to 66977
Data columns (total 7 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   k       66978 non-null  float64
 1   theta   66978 non-null  float64
 2   sigma   66978 non-null  float64
 3   v0      66978 non-null  float64
 4   rho     66978 non-null  float64
 5   T       66978 non-null  float64
 6   Strike  66978 non-null  float64
dtypes: float64(7)
memory usage: 3.6 MB
None
Head
          k     theta     sigma        v0       rho         T    Strike
0  0.509975  0.130755  0.139814  0.778713 -0.505420  0.687304  1.077324
1  0.571375  0.446076  0.937467  0.768080 -0.720002  0.672622  0.855332
Tail
              k     theta     sigma        v0       rho         T    Strike
66976  0.638318  0.716019  0.547159  0.649588 -0.987530  1.653131  0.667748
66977  0.823250  0.641822  0.722013  0.063890 -0.466869  1.975189  0.882732
Describe
                  k         theta         sigma            v0           rho  \
count  66978.000000  66978.000000  66978.000000  66978.000000  66978.000000
mean       0.504160      0.405256      0.505521      0.404513     -0.495030
```

```
std        0.285571      0.228084      0.286229      0.227821      0.286019
min        0.010005      0.010012      0.010005      0.010004     -0.989995
25%        0.257312      0.207648      0.256993      0.207200     -0.742858
50%        0.504307      0.405288      0.506148      0.404360     -0.494119
75%        0.750879      0.602834      0.753561      0.601198     -0.246403
max        0.999975      0.799988      0.999995      0.799996     -0.000015

                  T        Strike
count  66978.000000  66978.000000
mean       1.041012      0.999249
std        0.553530      0.230851
min        0.083362      0.600020
25%        0.560004      0.799014
50%        1.040469      0.997960
75%        1.521260      1.198882
max        1.999990      1.399972
```

1.2 Auxiliary Functions

```
In [69]: def show_scattered( y, t, tag, ax = None):
             #x       = model.predict(X)
             #y       = np.ravel(x)
             xMin = min(t)
             xMax = max(t)
             v    = np.arange(xMin, xMax, (xMax-xMin)/100.)

             diff   = np.fabs(y - t)
             print("@ %-24s: E[y-t]: %.6f Std(y-t): %.6f" %( tag, np.mean(diff), np.std(diff)))
             if ax == None: return

             ax.plot( y, t, ".")
             ax.plot( v, v, color="red")
             ax.set_title("%s mae=%8.4f, std=%8.4f" %(tag, np.mean(diff), np.std(diff)))
             ax.set_xlabel("predicted")
             ax.set_ylabel("target")
```

```
In [70]: def display_nn_results( model, X_train, X_test, t_train, t_test, resFile=None):

             fig, ax = plt.subplots(1,2, figsize=(12,6))
             fig.suptitle("Scattered plots")

             y_train  = np.ravel(model.predict(X_train))
             show_scattered( y_train, t_train, "InSample", ax = ax[0])

             diff   = np.fabs(y_train - t_train)
             RES    = pd.DataFrame({"predicted": y_train, "target": t_train, "err:": diff})
             RES.to_csv("res_in__sample.csv", sep=',', float_format="%.6f", index=True)
```

```
    print("@")
    y_test  = np.ravel(model.predict(X_test))
    show_scattered( y_test , t_test, "OutOfSample", ax= ax[1])

    diff   = np.fabs(y_test-t_test)
    RES    = pd.DataFrame({"predicted": y_test, "target": t_test, "err:": diff})
    RES.to_csv("res_out_sample.csv", sep=',', float_format="%.6f", index=True)

    print("@")

    if resFile != None:
        plt.savefig(resFile, format="png")
        print("@ %-12s: results saved to '%s' "%("Info", resFile))
    plt.show()


    score = model.evaluate(X_test, t_test, verbose=1)
    print('Score:'); print(score)
```

1.3 Build the model

```
In [71]: from keras.models import Sequential
         from keras.layers import Dense

         def model_builder( inputShape = (1,)):

             # Initialize the constructor
             model = Sequential()

             # Add an input layer
             #model.add(Dense(64, activation='relu', input_shape=(nrInputNodes,)))
             model.add(Dense(64, activation='relu', input_shape=inputShape))

             # Add one more hidden layer
             model.add(Dense(32, activation='relu'))

             # Add one more hidden layer
             model.add(Dense(16, activation='relu'))

             # Add an output layer
             model.add(Dense(1))
             # End model construction

             # Model output shape
             print("model.output_shape: %s" %(str(model.output_shape)))

             # Model summary
             print("Model.summary"); model.summary()
```

```python
    # Model config
    print("Model.config"); model.get_config()

    model.compile(loss='mse', optimizer='rmsprop', metrics=['mae'])
    return model
```

Let's go through this code line by line:

- The first line creates a **Sequential** model. This is the simplest kind of Keras model, for neural networks that are just composed of a single stack of layers, connected sequentially. This is called the sequential API.
- Next, we build the first layer and add it to the model. It is a **Dense** hidden layer with XXX neurons. It will use the **ReLu** activation function. Each Dense layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs. It also manages a vector of bias terms (one per neuron).
- Next we add a second Dense hidden layer with XXX neurons, also using the ReLu activation function and a third one …
- Finally, we add a Dense output layer with only 1 neurons, using the ReLu activation function (because…).

```python
In [72]: model = model_builder( inputShape = (X_train.shape[1],))
```

```
model.output_shape: (None, 1)
Model.summary
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 64)                512
_________________________________________________________________
dense_9 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_10 (Dense)             (None, 16)                528
_________________________________________________________________
dense_11 (Dense)             (None, 1)                 17
=================================================================
Total params: 3,137
Trainable params: 3,137
Non-trainable params: 0
_________________________________________________________________
Model.config
```

Note that Dense layers often have a lot of parameters. For example, the first hidden layer has n $\times$ n connection weights, plus 300 bias terms, which adds up to XXX parameters! This gives the model quite a lot of flexibility to fit the training data, but it also means that the model runs the risk of overfitting, especially when you do not have a lot of training data.

You can easily get a model's list of layers, to fetch a layer by its index, or you can fetch it by name:

```
In [73]: model.layers
```

```
Out[73]: [<tensorflow.python.keras.layers.core.Dense at 0x220a2bcb288>,
          <tensorflow.python.keras.layers.core.Dense at 0x2209c2417c8>,
          <tensorflow.python.keras.layers.core.Dense at 0x220a2be0b88>,
          <tensorflow.python.keras.layers.core.Dense at 0x2209c1f5d88>]
```

```
In [74]: model.layers[1].name
```

```
Out[74]: 'dense_9'
```

After a model is created, you must call its ***compile()*** method to specify the loss function and the optimizer to use. Optionally, you can also specify a list of extra metrics to compute during training and evaluation. In this case we have chosen

```
model.compile(loss='mse', optimizer='rmsprop', metrics=['mae'])
```

1.4 Train the model

Now the model is ready to be trained. For this we simply need to call its ***fit()*** method. We pass it the input features (***X_train***) and the target classes (***y_train***), as well as the number of epochs to train (or else it would default to just 1, which would definitely not be enough to converge to a good solution). We could also pass a validation set (this is optional): Keras will measure the loss and the extra metrics on this set at the end of each epoch, which is very useful to see how well the model really performs: if the performance on the training set is much better than on the validation set, your model is probably overfitting the training set (or there is a bug, such as a data mismatch between the training set and the validation set).

```
In [75]: history = model.fit(X_train, t_train, epochs=50, verbose=verbose)
```

```
Epoch 1/50
1403/1403 [==============================] - 2s 971us/step - loss: 0.0056 - mae: 0.0392
Epoch 2/50
1403/1403 [==============================] - 1s 958us/step - loss: 2.2207e-04 - mae: 0.0116
Epoch 3/50
1403/1403 [==============================] - 1s 988us/step - loss: 1.5558e-04 - mae: 0.0098
Epoch 4/50
1403/1403 [==============================] - 1s 944us/step - loss: 1.2956e-04 - mae: 0.0090
Epoch 5/50
1403/1403 [==============================] - 1s 951us/step - loss: 1.1956e-04 - mae: 0.0086
Epoch 6/50
1403/1403 [==============================] - 1s 960us/step - loss: 1.0862e-04 - mae: 0.0082
Epoch 7/50
1403/1403 [==============================] - 1s 963us/step - loss: 1.0330e-04 - mae: 0.0080
Epoch 8/50
```

```
1403/1403 [==============================] - 1s 1ms/step - loss: 1.0011e-04 - mae: 0.0079
Epoch 9/50
1403/1403 [==============================] - 1s 1ms/step - loss: 9.5863e-05 - mae: 0.0077
Epoch 10/50
1403/1403 [==============================] - 1s 1ms/step - loss: 9.3534e-05 - mae: 0.0077
Epoch 11/50
1403/1403 [==============================] - 1s 983us/step - loss: 9.2183e-05 - mae: 0.0076
Epoch 12/50
1403/1403 [==============================] - 1s 946us/step - loss: 8.8301e-05 - mae: 0.0074
Epoch 13/50
1403/1403 [==============================] - 1s 1ms/step - loss: 8.6879e-05 - mae: 0.0074A: 0s -
Epoch 14/50
1403/1403 [==============================] - 2s 1ms/step - loss: 8.5307e-05 - mae: 0.0073
Epoch 15/50
1403/1403 [==============================] - 1s 1ms/step - loss: 8.4984e-05 - mae: 0.0073
Epoch 16/50
1403/1403 [==============================] - 1s 1ms/step - loss: 8.4152e-05 - mae: 0.0072A: 0s -
Epoch 17/50
1403/1403 [==============================] - 1s 1ms/step - loss: 8.2589e-05 - mae: 0.0072A: 0s -
Epoch 18/50
1403/1403 [==============================] - 2s 1ms/step - loss: 8.1552e-05 - mae: 0.0072
Epoch 19/50
1403/1403 [==============================] - 2s 1ms/step - loss: 8.0461e-05 - mae: 0.0071
Epoch 20/50
1403/1403 [==============================] - 2s 1ms/step - loss: 7.9474e-05 - mae: 0.0071A: 0s -
Epoch 21/50
1403/1403 [==============================] - 2s 1ms/step - loss: 8.0504e-05 - mae: 0.0071
Epoch 22/50
1403/1403 [==============================] - 1s 971us/step - loss: 7.8534e-05 - mae: 0.0070 1s -
Epoch 23/50
1403/1403 [==============================] - 1s 939us/step - loss: 7.8302e-05 - mae: 0.0070
Epoch 24/50
1403/1403 [==============================] - 1s 943us/step - loss: 7.5989e-05 - mae: 0.0069
Epoch 25/50
1403/1403 [==============================] - 1s 963us/step - loss: 7.6612e-05 - mae: 0.0070
Epoch 26/50
1403/1403 [==============================] - 2s 1ms/step - loss: 7.5614e-05 - mae: 0.0069
Epoch 27/50
1403/1403 [==============================] - 2s 1ms/step - loss: 7.6360e-05 - mae: 0.0069
Epoch 28/50
1403/1403 [==============================] - 2s 1ms/step - loss: 7.5209e-05 - mae: 0.0069A: 1s -
Epoch 29/50
1403/1403 [==============================] - 2s 1ms/step - loss: 7.4739e-05 - mae: 0.0069
Epoch 30/50
1403/1403 [==============================] - 2s 2ms/step - loss: 7.5230e-05 - mae: 0.0069
Epoch 31/50
1403/1403 [==============================] - 2s 2ms/step - loss: 7.4841e-05 - mae: 0.0069A: 0s -
Epoch 32/50
```

```
1403/1403 [==============================] - 2s 1ms/step - loss: 7.3653e-05 - mae: 0.0068
Epoch 33/50
1403/1403 [==============================] - 1s 1000us/step - loss: 7.4516e-05 - mae: 0.0069
Epoch 34/50
1403/1403 [==============================] - 1s 994us/step - loss: 7.4291e-05 - mae: 0.0069 1s -
Epoch 35/50
1403/1403 [==============================] - 1s 931us/step - loss: 7.1987e-05 - mae: 0.0067
Epoch 36/50
1403/1403 [==============================] - 1s 938us/step - loss: 7.3242e-05 - mae: 0.0068
Epoch 37/50
1403/1403 [==============================] - 1s 945us/step - loss: 7.3535e-05 - mae: 0.0068 0s -
Epoch 38/50
1403/1403 [==============================] - 1s 939us/step - loss: 7.3011e-05 - mae: 0.0068
Epoch 39/50
1403/1403 [==============================] - 1s 936us/step - loss: 7.3236e-05 - mae: 0.0068
Epoch 40/50
1403/1403 [==============================] - 1s 951us/step - loss: 7.2252e-05 - mae: 0.0068
Epoch 41/50
1403/1403 [==============================] - 1s 963us/step - loss: 7.2084e-05 - mae: 0.0067
Epoch 42/50
1403/1403 [==============================] - 1s 954us/step - loss: 7.1792e-05 - mae: 0.0067
Epoch 43/50
1403/1403 [==============================] - 1s 956us/step - loss: 7.2343e-05 - mae: 0.0068
Epoch 44/50
1403/1403 [==============================] - 1s 946us/step - loss: 7.1964e-05 - mae: 0.0068
Epoch 45/50
1403/1403 [==============================] - 1s 973us/step - loss: 7.0856e-05 - mae: 0.0067
Epoch 46/50
1403/1403 [==============================] - 1s 964us/step - loss: 7.0766e-05 - mae: 0.0067
Epoch 47/50
1403/1403 [==============================] - 1s 960us/step - loss: 7.1087e-05 - mae: 0.0067
Epoch 48/50
1403/1403 [==============================] - 1s 964us/step - loss: 6.9860e-05 - mae: 0.0067
Epoch 49/50
1403/1403 [==============================] - 1s 983us/step - loss: 7.0814e-05 - mae: 0.0067
Epoch 50/50
1403/1403 [==============================] - 1s 1ms/step - loss: 6.9901e-05 - mae: 0.0066
```

And that's it! The neural network is trained. At each epoch during training, Keras displays the number of instances processed so far (along with a progress bar), the mean training time per sample, the loss and accuracy (or any other extra metrics you asked for), both on the training set and the validation set. You can see that the training loss went down, which is a good sign, and the validation accuracy reached XXX% after 50 epochs, not too far from the training accuracy, so there does not seem to be much overfitting going on.

All the parameters of a layer can be accessed using its ***get_weights()*** and ***set_weights()*** method. For a Dense layer, this includes both the connection weights and the bias terms:

```
In [76]: weights, biases = model.layers[1].get_weights()
```

```
weights
```

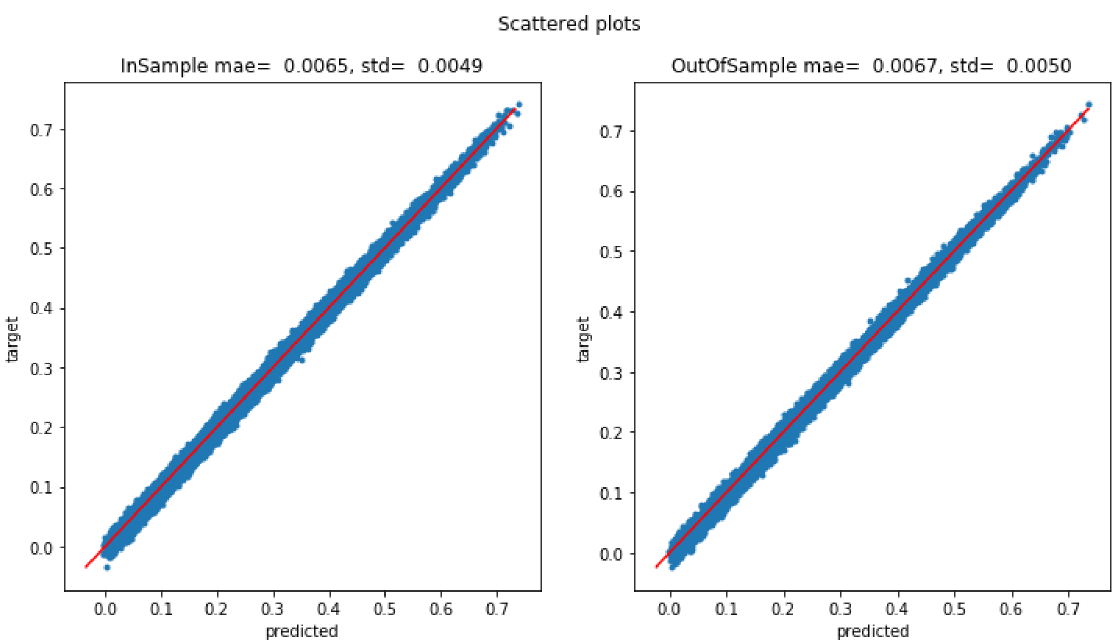
```
Out[76]: array([[-0.14440078,  0.10283986, -0.48962992, ..., -0.2326228 ,
                  0.08628539, -0.2668593 ],
                [ 0.01691153,  0.08426262,  0.1097483 , ...,  0.08745633,
                  0.00307687, -0.09567162],
                [-0.13278344,  0.21738702,  0.03779462, ..., -0.04654088,
                  0.33150652,  0.4644078 ],
                ...,
                [-0.281206  , -0.23986287,  0.42698872, ..., -0.21817678,
                 -0.04576546,  0.27727437],
                [-0.0995346 ,  0.02229814, -0.15240346, ..., -0.31588212,
                 -0.21563326, -0.2402453 ],
                [-0.14164002, -0.25760442,  0.13360217, ..., -0.23527765,
                 -0.20983057, -0.65304995]], dtype=float32)
```

```
In [77]: import warnings
         warnings.simplefilter('ignore')

         import matplotlib.pyplot as plt
         import numpy             as np

         display_nn_results(model, X_train, X_test, t_train, t_test, resFile = resFile)
```

```
@ InSample                : E[y-t]: 0.006511 Std(y-t): 0.004925
@
@ OutOfSample             : E[y-t]: 0.006693 Std(y-t): 0.005038
@
@ Info        : results saved to 'full_0000_trained.png'
```



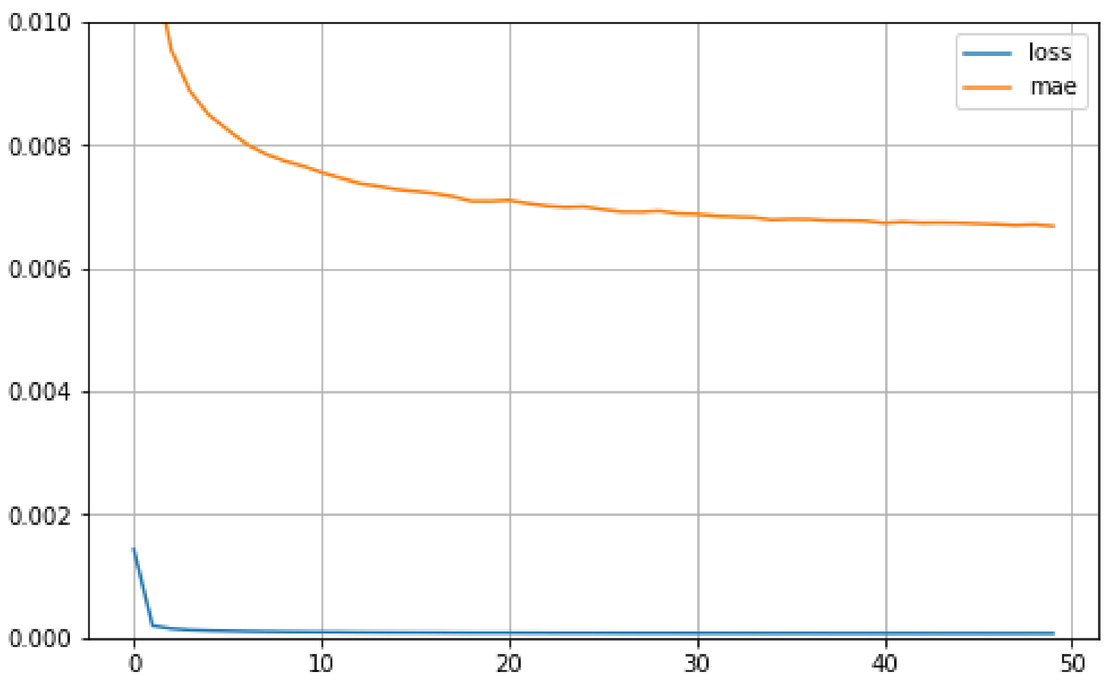
```
691/691 [==============================] - 1s 1ms/step - loss: 7.0181e-05 - mae: 0.0067A: 0s - l
Score:
[7.018065662123263e-05, 0.006692891474813223]
```

The ***fit()*** method returns a History object containing the training parameters (his tory.params), the list of epochs it went through (history.epoch), and most importantly a dictionary (history.history) containing the loss and extra metrics it measured at the end of each epoch on the training set and on the validation set (if any). If you create a Pandas DataFrame using this dictionary and call its plot() method, you get the learning curves shown in . . .

```
In [78]: import pandas as pd

         hist = pd.DataFrame(history.history)

         hist.plot(figsize=(8, 5))
         plt.grid(True)
         plt.gca().set_ylim(0, 0.01) # set the vertical range to [0-1]
         plt.show()
```




```
In [79]: hist.head()
```

```
Out[79]:        loss       mae
         0  0.001433  0.021834
         1  0.000198  0.011031
         2  0.000148  0.009540
         3  0.000126  0.008875
         4  0.000116  0.008491
```

1.5 Save Scaler and Model on Disk

```
In [80]: from pickle import dump, load

         dump(scaler, open(scalerFile, 'wb'))
         print("@ %-24s: scaler saved to '%s'" %("Info", scalerFile))

         model.save(mdlDir)
         print("@ %-24s: model saved to '%s'" %("Info", mdlDir))
```

```
@ Info                    : scaler saved to 'scaler_0000.pkl'
INFO:tensorflow:Assets written to: model_0000.krs\assets
@ Info                    : model saved to 'model_0000.krs'
```

```
In [ ]:
```