Part-D

June 20, 2020

1 Peer-graded Assignment: Build a Regression Model in Keras

1.0.1 Date: 20-June-2020

1.1 Download and Clean Dataset

[1]:
```
import pandas as pd
import numpy as np
```

We will be playing around with the same dataset that we used in the videos.

The dataset is about the compressive strength of different samples of concrete based on the volumes of the different ingredients that were used to make them. Ingredients include:

1.1.1 1. Cemen

1.1.2 2. Blast Furnace Slag

1.1.3 3. Fly Ash

1.1.4 4. Water

1.1.5 5. Superplasticizer

1.1.6 6. Coarse Aggregate

1.1.7 7. Fine Aggregate

Let's download the data and read it into a pandas dataframe.

[2]:
```
concrete_data = pd.read_csv('https://s3-api.us-geo.objectstorage.softlayer.net/
↪cf-courses-data/CognitiveClass/DL0101EN/labs/data/concrete_data.csv')
concrete_data.head(7)
```

[2]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	\
0	540.0	0.0	0.0	162.0	2.5	
1	540.0	0.0	0.0	162.0	2.5	
2	332.5	142.5	0.0	228.0	0.0	
3	332.5	142.5	0.0	228.0	0.0	
4	198.6	132.4	0.0	192.0	0.0	
5	266.0	114.0	0.0	228.0	0.0	
6	380.0	95.0	0.0	228.0	0.0	

```
   Coarse Aggregate  Fine Aggregate  Age  Strength
0            1040.0           676.0   28     79.99
1            1055.0           676.0   28     61.89
2             932.0           594.0  270     40.27
3             932.0           594.0  365     41.05
4             978.4           825.5  360     44.30
5             932.0           670.0   90     47.03
6             932.0           594.0  365     43.70
```

Let's check how many data points we have.

[3]:
```
concrete_data.shape
```

[3]:
```
(1030, 9)
```

Let's check the dataset for any missing values.

[4]:
```
concrete_data.describe()
```

[4]:
```
            Cement  Blast Furnace Slag      Fly Ash        Water  \
count  1030.000000         1030.000000  1030.000000  1030.000000
mean    281.167864           73.895825    54.188350   181.567282
std     104.506364           86.279342    63.997004    21.354219
min     102.000000            0.000000     0.000000   121.800000
25%     192.375000            0.000000     0.000000   164.900000
50%     272.900000           22.000000     0.000000   185.000000
75%     350.000000          142.950000   118.300000   192.000000
max     540.000000          359.400000   200.100000   247.000000

       Superplasticizer  Coarse Aggregate  Fine Aggregate          Age  \
count       1030.000000       1030.000000     1030.000000  1030.000000
mean           6.204660        972.918932      773.580485    45.662136
std            5.973841         77.753954       80.175980    63.169912
min            0.000000        801.000000      594.000000     1.000000
25%            0.000000        932.000000      730.950000     7.000000
50%            6.400000        968.000000      779.500000    28.000000
75%           10.200000       1029.400000      824.000000    56.000000
max           32.200000       1145.000000      992.600000   365.000000

          Strength
count  1030.000000
mean     35.817961
std      16.705742
min       2.330000
25%      23.710000
50%      34.445000
75%      46.135000
```

```
max      82.600000
```

[5]:
```
concrete_data.isnull().sum()
```

[5]:
```
Cement                0
Blast Furnace Slag    0
Fly Ash               0
Water                 0
Superplasticizer      0
Coarse Aggregate      0
Fine Aggregate        0
Age                   0
Strength              0
dtype: int64
```

The data looks very clean and is ready to be used to build our model.

Split data into predictors and target

[6]:
```
concrete_data_columns = concrete_data.columns

predictors = concrete_data[concrete_data_columns[concrete_data_columns !=␣
 ↪'Strength']] # all columns except Strength
target = concrete_data['Strength'] # Strength column
```

Let's do a quick sanity check of the predictors and the target dataframes.

[7]:
```
predictors.head(7)
```

[7]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age
0	540.0	0.0	0.0	162.0	2.5	1040.0	676.0	28
1	540.0	0.0	0.0	162.0	2.5	1055.0	676.0	28
2	332.5	142.5	0.0	228.0	0.0	932.0	594.0	270
3	332.5	142.5	0.0	228.0	0.0	932.0	594.0	365
4	198.6	132.4	0.0	192.0	0.0	978.4	825.5	360
5	266.0	114.0	0.0	228.0	0.0	932.0	670.0	90
6	380.0	95.0	0.0	228.0	0.0	932.0	594.0	365

[8]:
```
target.head(7)
```

[8]:
```
0    79.99
1    61.89
2    40.27
3    41.05
4    44.30
5    47.03
6    43.70
Name: Strength, dtype: float64
```

[9]:
```
predictors_norm = (predictors - predictors.mean()) / predictors.std()
predictors_norm.head(7)
```

[9]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age
0	2.476712	-0.856472	-0.846733	-0.916319	-0.620147	0.862735	-1.217079	-0.279597
1	2.476712	-0.856472	-0.846733	-0.916319	-0.620147	1.055651	-1.217079	-0.279597
2	0.491187	0.795140	-0.846733	2.174405	-1.038638	-0.526262	-2.239829	3.551340
3	0.491187	0.795140	-0.846733	2.174405	-1.038638	-0.526262	-2.239829	5.055221
4	-0.790075	0.678079	-0.846733	0.488555	-1.038638	0.070492	0.647569	4.976069
5	-0.145138	0.464818	-0.846733	2.174405	-1.038638	-0.526262	-1.291914	0.701883
6	0.945704	0.244603	-0.846733	2.174405	-1.038638	-0.526262	-2.239829	5.055221

Let's save the number of predictors to n_cols since we will need this number when building our network.

[10]:
```
n_cols = predictors_norm.shape[1] # number of predictors
n_cols
```

[10]:
```
8
```

1.2 Import Keras

Let's go ahead and import the Keras library

[11]:
```
import keras
```

```
Using TensorFlow backend.
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:519: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
```

```
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:520: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:521: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:522: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:523: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:528: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

Let's import the rest of the packages from the Keras library that we will need to build our regressoin model.

[12]:
```
from keras.models import Sequential
from keras.layers import Dense
```

1.2.1 Build a Neural Network

1.3 D. Increase the number of hidden layers

Repeat part B but use a neural network with the following instead:

Three hidden layers, each of 10 nodes and ReLU activation function.

[14]:
```
# define regression model
def regression_model():
    # create model
    model = Sequential()
    model.add(Dense(10, activation='relu', input_shape=(n_cols,)))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(10, activation='relu'))
```

```
    model.add(Dense(1))

    # compile model
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model
```

The above function creates a model that has one hidden layer with 10 neurons and a ReLU activation function. It uses the adam optimizer and the mean squared error as the loss function

1.4 Now we are going to, Repeat Part A but use a normalized version of the data.

1.4.1 Recall that one way to normalize the data is by subtracting the mean from the individual predictors and dividing by the standard deviation.

1. Randomly split the data into a training and test sets by holding 30% of the data for testing.

[15]:
```
from sklearn.model_selection import train_test_split
```

By using the train_test_split helper function from Scikit-learn.

[16]:
```
X_train, X_test, y_train, y_test = train_test_split(predictors_norm, target,␣
↪test_size=0.3, random_state=42)
```

1.5 Train and Test the Network

Let's call the function now to create our model.

[17]:
```
# build the model
model = regression_model()
```

2. Train the model on the training data using 50 epochs.

[18]:
```
# fit the model
model.fit(X_train, y_train, epochs=50, verbose=1)
```

```
Epoch 1/50
721/721 [==============================] - 2s 3ms/step - loss: 1591.8170
Epoch 2/50
721/721 [==============================] - 1s 805us/step - loss: 1577.1790
Epoch 3/50
721/721 [==============================] - 1s 943us/step - loss: 1559.9444
Epoch 4/50
721/721 [==============================] - 1s 1ms/step - loss: 1533.0120
Epoch 5/50
721/721 [==============================] - 1s 2ms/step - loss: 1493.3312
Epoch 6/50
721/721 [==============================] - 1s 2ms/step - loss: 1431.2451
Epoch 7/50
```

```
721/721 [==============================] - 1s 922us/step - loss: 1330.9075
Epoch 8/50
721/721 [==============================] - 1s 1ms/step - loss: 1164.5583
Epoch 9/50
721/721 [==============================] - 1s 1ms/step - loss: 928.2296
Epoch 10/50
721/721 [==============================] - 1s 995us/step - loss: 647.7081
Epoch 11/50
721/721 [==============================] - 1s 1ms/step - loss: 408.5308
Epoch 12/50
721/721 [==============================] - 1s 1ms/step - loss: 279.0268
Epoch 13/50
721/721 [==============================] - 1s 1ms/step - loss: 227.1808
Epoch 14/50
721/721 [==============================] - 1s 859us/step - loss: 201.0299
Epoch 15/50
721/721 [==============================] - 1s 1ms/step - loss: 185.3421
Epoch 16/50
721/721 [==============================] - 1s 1ms/step - loss: 174.8483A: 0s -
loss
Epoch 17/50
721/721 [==============================] - 1s 886us/step - loss: 167.0059
Epoch 18/50
721/721 [==============================] - ETA: 0s - loss: 162.506 - 1s 1ms/step
- loss: 162.1595
Epoch 19/50
721/721 [==============================] - 1s 1ms/step - loss: 157.7411
Epoch 20/50
721/721 [==============================] - 1s 1ms/step - loss: 154.6607
Epoch 21/50
721/721 [==============================] - 1s 1ms/step - loss: 151.7019
Epoch 22/50
721/721 [==============================] - 1s 1ms/step - loss: 149.3934
Epoch 23/50
721/721 [==============================] - 1s 940us/step - loss: 147.1737
Epoch 24/50
721/721 [==============================] - 1s 1ms/step - loss: 145.0226
Epoch 25/50
721/721 [==============================] - 1s 1ms/step - loss: 143.2988
Epoch 26/50
721/721 [==============================] - 1s 1ms/step - loss: 141.7003
Epoch 27/50
721/721 [==============================] - 1s 903us/step - loss: 140.0096 0s -
loss: 139.624
Epoch 28/50
721/721 [==============================] - 1s 1ms/step - loss: 138.7467
Epoch 29/50
128/721 [====>…] - ETA: 1s - loss: 134.6157
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/keras/callbacks.py:120: UserWarning: Method on_batch_end() is slow
compared to the batch update (0.160099). Check your callbacks.
  % delta_t_median)
721/721 [==============================] - 1s 2ms/step - loss: 137.0483
Epoch 30/50
721/721 [==============================] - 1s 1ms/step - loss: 135.9080
Epoch 31/50
721/721 [==============================] - 1s 925us/step - loss: 134.3764
Epoch 32/50
721/721 [==============================] - 1s 1ms/step - loss: 133.3511
Epoch 33/50
721/721 [==============================] - 1s 1ms/step - loss: 132.5623
Epoch 34/50
721/721 [==============================] - 1s 2ms/step - loss: 131.2262
Epoch 35/50
721/721 [==============================] - 1s 829us/step - loss: 130.4616 0s -
loss: 146.8 - ETA: 0s - loss: 129
Epoch 36/50
721/721 [==============================] - 1s 1ms/step - loss: 129.5860
Epoch 37/50
721/721 [==============================] - 1s 971us/step - loss: 128.6276
Epoch 38/50
721/721 [==============================] - 1s 1ms/step - loss: 127.8849
Epoch 39/50
721/721 [==============================] - 1s 998us/step - loss: 126.8479
Epoch 40/50
721/721 [==============================] - 1s 1ms/step - loss: 126.1915A: 0s -
loss:
Epoch 41/50
721/721 [==============================] - 1s 880us/step - loss: 125.5186
Epoch 42/50
721/721 [==============================] - 1s 1ms/step - loss: 124.6117
Epoch 43/50
721/721 [==============================] - 1s 1ms/step - loss: 124.0102
Epoch 44/50
721/721 [==============================] - 1s 1ms/step - loss: 123.4994
Epoch 45/50
721/721 [==============================] - 1s 1ms/step - loss: 122.5946
Epoch 46/50
721/721 [==============================] - 1s 922us/step - loss: 121.6403
Epoch 47/50
721/721 [==============================] - 1s 1ms/step - loss: 120.9717
Epoch 48/50
721/721 [==============================] - 1s 1ms/step - loss: 120.2223
Epoch 49/50
721/721 [==============================] - 1s 942us/step - loss: 119.6901
```

```
Epoch 50/50
721/721 [==============================] - 1s 885us/step - loss: 119.1876
```

[18]:
```
<keras.callbacks.History at 0x7f3d6b501438>
```

3a. Evaluate the model on the test data.

[19]:
```
loss_val = model.evaluate(X_test, y_test)
y_pred = model.predict(X_test)
loss_val
```

```
309/309 [==============================] - 0s 901us/step
```

[19]:
```
125.23177490728187
```

3b. And now we compute the mean squared error between the predicted concrete strength and the actual concrete strength.

You can use the mean_squared_error function from Scikit-learn.

[20]:
```
from sklearn.metrics import mean_squared_error
```

[21]:
```
mean_square_error = mean_squared_error(y_test, y_pred)
mean = np.mean(mean_square_error)

standard_deviation = np.std(mean_square_error)

print (mean, standard_deviation)
```

```
125.2317779610966 0.0
```

4. Repeat steps 1 - 3, 50 times, i.e., create a list of 50 mean squared errors.

[22]:
```
# To Repeat 50 Times
total_mean_squared_error = 50

mean_squared_errors = []

for i in range(0, total_mean_squared_error):
    X_train, X_test, y_train, y_test = train_test_split(predictors_norm,
                                                        target,
                                                        test_size=0.3,
                                                        random_state=i)
    model.fit(X_train, y_train, epochs = 50, verbose = 0)
    MSE = model.evaluate(X_test, y_test, verbose = 0)
    print ("MSE " + str(i + 1)+" : " + str(MSE))

    y_pred = model.predict(X_test)
```

```
    mean_square_error = mean_squared_error(y_test, y_pred)
    mean_squared_errors.append(mean_square_error)
```

```
MSE 1 : 92.70640959014399
MSE 2 : 60.47892015574433
MSE 3 : 43.6860404276925
MSE 4 : 44.27461982159167
MSE 5 : 43.96891151354151
MSE 6 : 41.828688785096205
MSE 7 : 46.690756393482
MSE 8 : 34.658195075865315
MSE 9 : 36.66403636500288
MSE 10 : 34.58553784564861
MSE 11 : 33.462948894809365
MSE 12 : 30.292143701349648
MSE 13 : 38.16285069166264
MSE 14 : 37.307800762090096
MSE 15 : 35.78732965216282
MSE 16 : 25.208672341405382
MSE 17 : 30.074034798878298
MSE 18 : 32.08739646121522
MSE 19 : 30.198655063666187
MSE 20 : 33.99533356354846
MSE 21 : 30.187741745637073
MSE 22 : 30.61458512957428
MSE 23 : 26.938009515163582
MSE 24 : 28.887424592447125
MSE 25 : 30.833555783268704
MSE 26 : 28.56393111639424
MSE 27 : 26.78158715010461
MSE 28 : 28.27740997635431
MSE 29 : 27.65398276122257
MSE 30 : 26.923859821554142
MSE 31 : 22.372154667925294
MSE 32 : 23.722599813467475
MSE 33 : 23.23853293977509
MSE 34 : 25.748447634255616
MSE 35 : 29.036564478210646
MSE 36 : 31.979924538374718
MSE 37 : 24.2809244081812
MSE 38 : 25.051702024095654
MSE 39 : 23.98979033930016
MSE 40 : 20.5914996187278
MSE 41 : 29.19130209888841
MSE 42 : 20.375381006777864
```

```
MSE 43 : 25.105287150657677
MSE 44 : 28.169573435119826
MSE 45 : 25.80984772370471
MSE 46 : 24.14206602966901
MSE 47 : 23.742543044599515
MSE 48 : 22.48302817730456
MSE 49 : 24.68799628100349
MSE 50 : 22.884506762606428
```

[24]:
```
mean_squared_errors = np.array(mean_squared_errors)

mean = np.mean(mean_squared_errors)

standard_deviation = np.std(mean_squared_errors)

print('\n')
print("Below is the mean and standard deviation of "␣
 ↪+str(total_mean_squared_error) + " mean squared errors with normalized data.␣
 ↪Total number of epochs used for each training is: 50" + "\n")
print("Mean: "+str(mean))
print("Standard Deviation: "+str(standard_deviation))
```

```
      ␣
↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
↪last)

       <ipython-input-24-3b3ea08f3984> in <module>
         6
         7 print('\n')
   ----> 8 print("Below is the mean and standard deviation of "␣
↪+str(total_mean_squared_errors) + " mean squared errors with normalized data.␣
↪Total number of epochs used for each training is: 50" + "\n")
         9 print("Mean: "+str(mean))
        10 print("Standard Deviation: "+str(standard_deviation))


       NameError: name 'total_mean_squared_errors' is not defined
```

[ ]:
```

```