Part-B

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()
```

[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

   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
```

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]:
```python
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]:
```python
predictors.head()
```

[7]:
```
   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

   Coarse Aggregate  Fine Aggregate  Age
0            1040.0           676.0   28
1            1055.0           676.0   28
2             932.0           594.0  270
3             932.0           594.0  365
4             978.4           825.5  360
```

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

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

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

[9]:
```
     Cement  Blast Furnace Slag   Fly Ash     Water  Superplasticizer  \
0  2.476712           -0.856472 -0.846733 -0.916319         -0.620147
1  2.476712           -0.856472 -0.846733 -0.916319         -0.620147
2  0.491187            0.795140 -0.846733  2.174405         -1.038638
3  0.491187            0.795140 -0.846733  2.174405         -1.038638
4 -0.790075            0.678079 -0.846733  0.488555         -1.038638

   Coarse Aggregate  Fine Aggregate       Age
0          0.862735       -1.217079 -0.279597
1          1.055651       -1.217079 -0.279597
2         -0.526262       -2.239829  3.551340
3         -0.526262       -2.239829  5.055221
4          0.070492        0.647569  4.976069
```

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

[10]:
```python
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]:
```python
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]:
```python
from keras.models import Sequential
from keras.layers import Dense
```

Build a Neural Network

[13]:
```python
# define regression model
def regression_model():
    # create model
    model = Sequential()
    model.add(Dense(10, activation='relu', input_shape=(n_cols,)))
    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

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

1.3.1 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.

[14]:
```python
from sklearn.model_selection import train_test_split
```

By using the train_test_split helper function from Scikit-learn.

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

1.4 Train and Test the Network

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

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

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

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

```
Epoch 1/50
721/721 [==============================] - 1s 806us/step - loss: 1634.2620
Epoch 2/50
721/721 [==============================] - 0s 333us/step - loss: 1621.4399
Epoch 3/50
721/721 [==============================] - 0s 414us/step - loss: 1609.9325
Epoch 4/50
721/721 [==============================] - 0s 337us/step - loss: 1599.0050
Epoch 5/50
721/721 [==============================] - 0s 527us/step - loss: 1588.7868
Epoch 6/50
721/721 [==============================] - 0s 578us/step - loss: 1579.1333
Epoch 7/50
721/721 [==============================] - 0s 469us/step - loss: 1569.3737
Epoch 8/50
721/721 [==============================] - 0s 556us/step - loss: 1559.50000s -
loss: 1559.84
Epoch 9/50
721/721 [==============================] - 0s 581us/step - loss: 1549.1868
Epoch 10/50
721/721 [==============================] - 0s 523us/step - loss: 1538.2371
Epoch 11/50
721/721 [==============================] - 0s 529us/step - loss: 1526.2811
Epoch 12/50
721/721 [==============================] - 0s 502us/step - loss: 1513.2514
Epoch 13/50
721/721 [==============================] - 0s 365us/step - loss: 1499.1205
Epoch 14/50
721/721 [==============================] - 0s 352us/step - loss: 1483.6528
Epoch 15/50
721/721 [==============================] - 0s 332us/step - loss: 1466.3393
Epoch 16/50
721/721 [==============================] - 0s 330us/step - loss: 1448.1687
```

```
Epoch 17/50
721/721 [==============================] - 0s 307us/step - loss: 1428.6539
Epoch 18/50
721/721 [==============================] - 0s 356us/step - loss: 1407.7848
Epoch 19/50
721/721 [==============================] - 0s 308us/step - loss: 1386.0323
Epoch 20/50
721/721 [==============================] - 0s 329us/step - loss: 1363.0622
Epoch 21/50
721/721 [==============================] - 0s 332us/step - loss: 1338.9986
Epoch 22/50
721/721 [==============================] - 0s 356us/step - loss: 1314.2821
Epoch 23/50
721/721 [==============================] - 0s 332us/step - loss: 1288.4328
Epoch 24/50
721/721 [==============================] - 0s 303us/step - loss: 1261.5362
Epoch 25/50
721/721 [==============================] - 0s 329us/step - loss: 1234.4014
Epoch 26/50
721/721 [==============================] - 0s 306us/step - loss: 1206.1250
Epoch 27/50
721/721 [==============================] - 0s 666us/step - loss: 1177.3462
Epoch 28/50
721/721 [==============================] - 0s 330us/step - loss: 1147.9137
Epoch 29/50
721/721 [==============================] - 0s 331us/step - loss: 1117.6303
Epoch 30/50
721/721 [==============================] - 0s 282us/step - loss: 1086.2562
Epoch 31/50
721/721 [==============================] - 0s 333us/step - loss: 1054.4222
Epoch 32/50
721/721 [==============================] - 0s 326us/step - loss: 1021.9631
Epoch 33/50
721/721 [==============================] - 0s 360us/step - loss: 989.4795
Epoch 34/50
721/721 [==============================] - 0s 304us/step - loss: 957.0086
Epoch 35/50
721/721 [==============================] - 0s 332us/step - loss: 924.4756
Epoch 36/50
721/721 [==============================] - 0s 328us/step - loss: 892.3896
Epoch 37/50
721/721 [==============================] - 0s 332us/step - loss: 860.9179
Epoch 38/50
721/721 [==============================] - 0s 332us/step - loss: 829.5048
Epoch 39/50
721/721 [==============================] - 0s 310us/step - loss: 799.0575
Epoch 40/50
721/721 [==============================] - 0s 325us/step - loss: 769.1396
```

```
Epoch 41/50
721/721 [==============================] - 0s 308us/step - loss: 739.8831
Epoch 42/50
721/721 [==============================] - 0s 353us/step - loss: 711.5372
Epoch 43/50
721/721 [==============================] - 0s 390us/step - loss: 684.2270
Epoch 44/50
721/721 [==============================] - 0s 328us/step - loss: 657.7836
Epoch 45/50
721/721 [==============================] - 0s 306us/step - loss: 632.1052
Epoch 46/50
721/721 [==============================] - 0s 312us/step - loss: 607.3903
Epoch 47/50
721/721 [==============================] - 0s 307us/step - loss: 583.9228
Epoch 48/50
721/721 [==============================] - 0s 358us/step - loss: 561.2839
Epoch 49/50
721/721 [==============================] - 0s 358us/step - loss: 539.7439
Epoch 50/50
721/721 [==============================] - 0s 308us/step - loss: 519.1636
```

[17]: <keras.callbacks.History at 0x7f8ace44fe48>

3a. Evaluate the model on the test data.

[18]:
```
loss_val = model.evaluate(X_test, y_test)
y_pred = model.predict(X_test)
loss_val
```

```
309/309 [==============================] - 0s 304us/step
```

[18]: 478.55955223898286

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.

[19]:
```
from sklearn.metrics import mean_squared_error
```

[20]:
```
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)
```

```
478.5595553559433 0.0
```

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

[21]:
```python
# 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 : 176.0293610211715
MSE 2 : 124.86968144784082
MSE 3 : 82.58524694103254
MSE 4 : 75.87992098416325
MSE 5 : 67.45893035197335
MSE 6 : 62.37414958176104
MSE 7 : 54.62136719836386
MSE 8 : 37.6529198557042
MSE 9 : 39.4175199737055
MSE 10 : 39.22335485810215
MSE 11 : 39.88129099132945
MSE 12 : 35.87162084486878
MSE 13 : 42.72135316052483
MSE 14 : 43.238072750252044
MSE 15 : 36.07689674155226
MSE 16 : 31.689901839568005
MSE 17 : 33.1054441211293
MSE 18 : 33.237843251151176
MSE 19 : 31.52707493112311
MSE 20 : 35.08619664633544
MSE 21 : 29.87743132014105
MSE 22 : 31.357572216046282
MSE 23 : 29.57729167999959
MSE 24 : 34.04183020483715
MSE 25 : 35.092334660897365
```

```
MSE 26 : 36.04648446882427
MSE 27 : 31.75002222153747
MSE 28 : 30.405525880338306
MSE 29 : 35.72229112544878
MSE 30 : 33.43627421061198
MSE 31 : 31.05564333474366
MSE 32 : 28.39397348940951
MSE 33 : 29.085554413039322
MSE 34 : 31.217179381731643
MSE 35 : 33.14380788957417
MSE 36 : 38.09117938168227
MSE 37 : 29.061464352900927
MSE 38 : 33.67532141231796
MSE 39 : 30.830174159077764
MSE 40 : 27.038878925409904
MSE 41 : 33.52155779866339
MSE 42 : 26.36180075858403
MSE 43 : 31.599976530352844
MSE 44 : 35.96960881304201
MSE 45 : 33.50802367867775
MSE 46 : 32.87414796452692
MSE 47 : 30.272057888191494
MSE 48 : 32.57862328094186
MSE 49 : 32.44731559876871
MSE 50 : 32.76418852574617
```

[24]:
```python
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))
```

```
Below is the mean and standard deviation of 50 mean squared errors with
normalized data. Total number of epochs used for each training is: 50

Mean: 41.66691388175126
Standard Deviation: 25.524449003341797
```

```
[ ]:
```