Part-C

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

[2]:
```
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.

[3]:
```
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()
```

[3]:
```
   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.

[4]:
```
concrete_data.shape
```

[4]:
```
(1030, 9)
```

Let's check the dataset for any missing values.

[5]:
```
concrete_data.describe()
```

[5]:
```
            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
```

[6]:
```
concrete_data.isnull().sum()
```

[6]:
```
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

[7]:
```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.

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

[8]:
```
   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
```

[9]:
```python
target.head()
```

[9]:
```
0    79.99
1    61.89
2    40.27
3    41.05
4    44.30
Name: Strength, dtype: float64
```

[10]:
```
predictors_norm = (predictors - predictors.mean()) / predictors.std()
predictors_norm.head()
```

[10]:
```
     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.

[11]:
```
n_cols = predictors_norm.shape[1] # number of predictors
n_cols
```

[11]: 8

1.2 Import Keras

Let's go ahead and import the Keras library

[12]:
```
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.

[14]:
```python
from keras.models import Sequential
from keras.layers import Dense
```

Build a Neural Network

[15]:
```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.

[16]:
```python
from sklearn.model_selection import train_test_split
```

By using the train_test_split helper function from Scikit-learn.

[17]:
```
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.

[18]:
```
# build the model
model = regression_model()
```

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

[19]:
```
# fit the model
model.fit(X_train, y_train, epochs=50, verbose=1)
```

```
Epoch 1/50
721/721 [==============================] - 1s 948us/step - loss: 1571.9921
Epoch 2/50
721/721 [==============================] - 0s 314us/step - loss: 1553.2947
Epoch 3/50
721/721 [==============================] - 0s 417us/step - loss: 1534.3422
Epoch 4/50
721/721 [==============================] - 0s 444us/step - loss: 1515.3604
Epoch 5/50
721/721 [==============================] - 0s 420us/step - loss: 1496.2630
Epoch 6/50
721/721 [==============================] - 0s 390us/step - loss: 1476.1291
Epoch 7/50
721/721 [==============================] - 0s 331us/step - loss: 1455.7080
Epoch 8/50
721/721 [==============================] - 0s 334us/step - loss: 1434.3188
Epoch 9/50
721/721 [==============================] - 0s 414us/step - loss: 1412.4823
Epoch 10/50
721/721 [==============================] - 0s 605us/step - loss: 1389.5835
Epoch 11/50
721/721 [==============================] - ETA: 0s - loss: 1377.82 - 0s
663us/step - loss: 1365.8016
Epoch 12/50
721/721 [==============================] - 0s 555us/step - loss: 1340.9070
Epoch 13/50
721/721 [==============================] - 0s 578us/step - loss: 1315.2953
Epoch 14/50
721/721 [==============================] - 0s 634us/step - loss: 1288.2201
Epoch 15/50
721/721 [==============================] - 0s 662us/step - loss: 1260.2255
Epoch 16/50
721/721 [==============================] - 0s 441us/step - loss: 1230.8268
```

```
Epoch 17/50
721/721 [==============================] - 0s 337us/step - loss: 1200.1475
Epoch 18/50
721/721 [==============================] - 0s 384us/step - loss: 1168.1549
Epoch 19/50
721/721 [==============================] - 0s 357us/step - loss: 1134.8187
Epoch 20/50
721/721 [==============================] - 0s 441us/step - loss: 1100.1094
Epoch 21/50
721/721 [==============================] - 0s 411us/step - loss: 1064.4192
Epoch 22/50
721/721 [==============================] - 0s 538us/step - loss: 1027.6713
Epoch 23/50
721/721 [==============================] - 0s 354us/step - loss: 990.5544
Epoch 24/50
721/721 [==============================] - 0s 360us/step - loss: 952.3005
Epoch 25/50
721/721 [==============================] - 0s 302us/step - loss: 913.9482
Epoch 26/50
721/721 [==============================] - 0s 310us/step - loss: 876.1267
Epoch 27/50
721/721 [==============================] - 0s 329us/step - loss: 838.2035
Epoch 28/50
721/721 [==============================] - 0s 328us/step - loss: 800.6341
Epoch 29/50
721/721 [==============================] - 0s 411us/step - loss: 763.8534
Epoch 30/50
721/721 [==============================] - 0s 324us/step - loss: 727.7134
Epoch 31/50
721/721 [==============================] - 0s 392us/step - loss: 692.5785
Epoch 32/50
721/721 [==============================] - 0s 333us/step - loss: 658.6326
Epoch 33/50
721/721 [==============================] - 0s 306us/step - loss: 626.0631
Epoch 34/50
721/721 [==============================] - 0s 335us/step - loss: 594.1754
Epoch 35/50
721/721 [==============================] - 0s 464us/step - loss: 564.9217
Epoch 36/50
721/721 [==============================] - 0s 359us/step - loss: 535.9718 0s -
loss: 561.43
Epoch 37/50
721/721 [==============================] - 0s 469us/step - loss: 509.4540
Epoch 38/50
721/721 [==============================] - 0s 360us/step - loss: 484.1051
Epoch 39/50
721/721 [==============================] - 0s 388us/step - loss: 460.3652
Epoch 40/50
```

```
721/721 [==============================] - 0s 420us/step - loss: 438.4018
Epoch 41/50
721/721 [==============================] - 0s 379us/step - loss: 418.5512
Epoch 42/50
721/721 [==============================] - 0s 332us/step - loss: 399.6722
Epoch 43/50
721/721 [==============================] - 0s 334us/step - loss: 382.3411
Epoch 44/50
721/721 [==============================] - 0s 443us/step - loss: 366.9112
Epoch 45/50
721/721 [==============================] - 0s 334us/step - loss: 352.0659
Epoch 46/50
721/721 [==============================] - 0s 333us/step - loss: 339.2806
Epoch 47/50
721/721 [==============================] - 0s 332us/step - loss: 327.0471
Epoch 48/50
721/721 [==============================] - 0s 357us/step - loss: 316.0737
Epoch 49/50
721/721 [==============================] - 0s 332us/step - loss: 306.1970
Epoch 50/50
721/721 [==============================] - 0s 466us/step - loss: 297.2147
```

[19]:
```
<keras.callbacks.History at 0x7fd5149af240>
```

3a. Evaluate the model on the test data.

[20]:
```python
loss_val = model.evaluate(X_test, y_test)
y_pred = model.predict(X_test)
loss_val
```

```
309/309 [==============================] - 0s 345us/step
```

[20]:
```
272.8632547816798
```

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.

[21]:
```python
from sklearn.metrics import mean_squared_error
```

[22]:
```python
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)
```

```
272.86325234095654 0.0
```

1.5 C. Increate the number of epochs

1.5.1 Repeat Part B but use 100 epochs this time for training.

4. Repeat steps 1 - 3, 100 times, i.e., create a list of 100 mean squared errors.

[23]:
```
# To Repeat 100 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 = 100, 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 : 110.99258158662172
MSE 2 : 88.40879458825565
MSE 3 : 58.04875532477419
MSE 4 : 47.233559870797066
MSE 5 : 44.238216535944765
MSE 6 : 45.93428970386295
MSE 7 : 45.81724037244482
MSE 8 : 33.75147116454288
MSE 9 : 37.615485219122135
MSE 10 : 38.18656937435607
MSE 11 : 38.78876153396557
MSE 12 : 34.18751129594821
MSE 13 : 42.59161163379459
MSE 14 : 42.374165852864586
MSE 15 : 35.76423572182269
MSE 16 : 32.28822811213126
MSE 17 : 37.74920812316697
MSE 18 : 37.25092526161169
MSE 19 : 35.76987774549565
MSE 20 : 36.402560558133914
MSE 21 : 34.1115195388547
MSE 22 : 35.171923449124336
```

```
MSE 23 : 30.39963526556021
MSE 24 : 35.08827748962205
MSE 25 : 37.77801910573225
MSE 26 : 35.528094739204086
MSE 27 : 33.19477791153497
MSE 28 : 33.5122469868089
MSE 29 : 39.21997183889247
MSE 30 : 37.12856014955391
MSE 31 : 33.13610828732981
MSE 32 : 33.88595458218966
MSE 33 : 33.55270767211914
MSE 34 : 35.63654124929681
MSE 35 : 34.9255379982365
MSE 36 : 42.68975767734367
MSE 37 : 30.87825617435295
MSE 38 : 37.47720781344812
MSE 39 : 35.919412915374856
MSE 40 : 30.945620725070004
MSE 41 : 37.0647648561348
MSE 42 : 30.66138106410943
MSE 43 : 36.197793756873864
MSE 44 : 37.65189624527126
MSE 45 : 36.70062643501751
MSE 46 : 37.620070139567055
MSE 47 : 34.35568496555958
MSE 48 : 37.08429135319484
MSE 49 : 34.85159885690436
MSE 50 : 38.41668404884709
```

[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: 100" + "\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: 100

Mean: 39.68357911045612
```

```
Standard Deviation: 13.359958486786619
```

[ ]:

```

```