regression model using the deep learning Keras library

April 28, 2020

1 Regression model using the deep learning Keras library

Objective: I am going to build a regression model using the Keras library to model the data about concrete compressive strength. -The predictors in the data of concrete strength include:

1. Cement
2. Blast Furnace Slag
3. Fly Ash
4. Water
5. Superplasticizer
6. Coarse Aggregate
7. Fine Aggregate

2 Download and Clean Dataset

Let's start by importing the pandas and the Numpy libraries.

[25]:
```
import pandas as pd
import numpy as np
```

[26]:
```
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()
```

[26]:
```
   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
```

[27]:
```
concrete_data.shape #data points
```

[27]:
```
(1030, 9)
```

[28]:
```
concrete_data.describe()
```

[28]:
```
            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
```

[29]:
```
concrete_data.isnull().sum() # CLEAN THE DATA
```

[29]:
```
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
```

2.0.1 Split data into predictors and target

[30]:
```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
```

[31]:
```python
predictors.head()
```

[31]:
```
   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
```

[32]:
```python
target.head()
```

[32]:
```
0    79.99
1    61.89
2    40.27
3    41.05
4    44.30
Name: Strength, dtype: float64
```

normalize the data by substracting the mean and dividing by the standard deviation.

[33]:
```python
predictors_norm = (predictors - predictors.mean()) / predictors.std()
predictors_norm.head()
```

[33]:
```
     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
```

2.0.2 save the number of predictors to n_cols

[34]:
```python
n_cols = predictors_norm.shape[1] # number of predictors
```

3 Import the Keras library

[35]:
```python
import keras
```

[36]:
```python
from keras.models import Sequential
from keras.layers import Dense
```

4 Build a Neural Network

4.1 Model that has one hidden layer with 20 neurons and a ReLU activation function. It uses the adam optimizer and the mean squared error as the loss function.

[37]:
```python
# define regression model
def regression_model():
    # create model
    model = Sequential()
    model.add(Dense(20, activation='relu', input_shape=(n_cols,)))
    model.add(Dense(20, activation='relu'))
    model.add(Dense(1))

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

4.1.1 import scikit-learn in order to randomly split the data into a training and test sets

[38]:
```python
from sklearn.model_selection import train_test_split
```

4.1.2 Splitting the data into a training and test sets by holding 30% of the data for testing

[39]:
```
X_train, X_test, y_train, y_test = train_test_split(predictors, target,␣
↪test_size=0.3, random_state=42)
```

5 Train and Test the Network

[40]:
```
# build the model
model = regression_model()
```

5.0.1 Next, we will train and test the model at the same time using the fit method. We will leave out 30% of the data for validation and we will train the model for 50 epochs.

[41]:
```
# fit the model
model.fit(predictors_norm, target, validation_split=0.3, epochs=50, verbose=1)
```

```
Train on 721 samples, validate on 309 samples
Epoch 1/50
721/721 [==============================] - 1s 1ms/step - loss: 1592.1813 -
val_loss: 1123.0400
Epoch 2/50
721/721 [==============================] - 0s 467us/step - loss: 1523.3531 -
val_loss: 1059.7397
Epoch 3/50
721/721 [==============================] - 0s 423us/step - loss: 1430.3689 -
val_loss: 977.5203
Epoch 4/50
721/721 [==============================] - 0s 436us/step - loss: 1305.3225 -
val_loss: 869.0520
Epoch 5/50
721/721 [==============================] - 0s 397us/step - loss: 1140.9002 -
val_loss: 734.6208
Epoch 6/50
721/721 [==============================] - 0s 353us/step - loss: 938.0093 -
val_loss: 583.1781
Epoch 7/50
721/721 [==============================] - 0s 411us/step - loss: 718.2597 -
val_loss: 432.2955
Epoch 8/50
721/721 [==============================] - 0s 414us/step - loss: 515.2361 -
val_loss: 313.2442
Epoch 9/50
721/721 [==============================] - 0s 443us/step - loss: 362.7380 -
val_loss: 239.4237
Epoch 10/50
```

```
721/721 [==============================] - 0s 420us/step - loss: 276.5404 -
val_loss: 204.7216
Epoch 11/50
721/721 [==============================] - 0s 412us/step - loss: 241.5603 -
val_loss: 188.8415
Epoch 12/50
721/721 [==============================] - 0s 444us/step - loss: 225.2368 -
val_loss: 184.7508
Epoch 13/50
721/721 [==============================] - 0s 418us/step - loss: 216.2540 -
val_loss: 179.9259
Epoch 14/50
721/721 [==============================] - 0s 391us/step - loss: 209.1906 -
val_loss: 177.9733
Epoch 15/50
721/721 [==============================] - 0s 442us/step - loss: 202.3530 -
val_loss: 176.2087
Epoch 16/50
721/721 [==============================] - 0s 413us/step - loss: 196.8938 -
val_loss: 173.6443
Epoch 17/50
721/721 [==============================] - 0s 385us/step - loss: 192.3092 -
val_loss: 171.2785
Epoch 18/50
721/721 [==============================] - 0s 392us/step - loss: 188.4346 -
val_loss: 171.0406
Epoch 19/50
721/721 [==============================] - 0s 386us/step - loss: 183.8704 -
val_loss: 167.8964
Epoch 20/50
721/721 [==============================] - 0s 385us/step - loss: 180.1507 -
val_loss: 166.9561
Epoch 21/50
721/721 [==============================] - 0s 418us/step - loss: 177.3921 -
val_loss: 166.5221
Epoch 22/50
721/721 [==============================] - 1s 773us/step - loss: 173.7074 -
val_loss: 163.9418
Epoch 23/50
721/721 [==============================] - 0s 415us/step - loss: 171.3992 -
val_loss: 162.7826
Epoch 24/50
721/721 [==============================] - 0s 490us/step - loss: 168.4965 -
val_loss: 163.0351
Epoch 25/50
721/721 [==============================] - 0s 447us/step - loss: 166.1308 -
val_loss: 162.3942
Epoch 26/50
```

```
721/721 [==============================] - 0s 446us/step - loss: 163.3589 -
val_loss: 161.7177
Epoch 27/50
721/721 [==============================] - 0s 444us/step - loss: 161.5190 -
val_loss: 160.9315
Epoch 28/50
721/721 [==============================] - 0s 461us/step - loss: 160.0235 -
val_loss: 159.6231
Epoch 29/50
721/721 [==============================] - 0s 367us/step - loss: 157.6023 -
val_loss: 159.4246
Epoch 30/50
721/721 [==============================] - 0s 414us/step - loss: 155.9189 -
val_loss: 158.4316
Epoch 31/50
721/721 [==============================] - 0s 409us/step - loss: 154.5417 -
val_loss: 159.1487
Epoch 32/50
721/721 [==============================] - 0s 503us/step - loss: 153.1457 -
val_loss: 159.0071
Epoch 33/50
721/721 [==============================] - 0s 415us/step - loss: 152.0043 -
val_loss: 158.7030
Epoch 34/50
721/721 [==============================] - 0s 447us/step - loss: 150.1376 -
val_loss: 160.7035
Epoch 35/50
721/721 [==============================] - 0s 550us/step - loss: 149.2504 -
val_loss: 159.8873
Epoch 36/50
721/721 [==============================] - 0s 447us/step - loss: 147.8266 -
val_loss: 158.5729
Epoch 37/50
721/721 [==============================] - 0s 416us/step - loss: 146.7138 -
val_loss: 158.9099
Epoch 38/50
721/721 [==============================] - 1s 747us/step - loss: 145.6125 -
val_loss: 158.9949
Epoch 39/50
721/721 [==============================] - 0s 478us/step - loss: 144.4776 -
val_loss: 158.7869
Epoch 40/50
721/721 [==============================] - 0s 391us/step - loss: 143.2285 -
val_loss: 159.1876
Epoch 41/50
721/721 [==============================] - 0s 343us/step - loss: 142.6029 -
val_loss: 160.4243
Epoch 42/50
```

```
721/721 [==============================] - 0s 342us/step - loss: 141.4230 -
val_loss: 159.6774
Epoch 43/50
721/721 [==============================] - 0s 413us/step - loss: 140.3492 -
val_loss: 159.8584
Epoch 44/50
721/721 [==============================] - 0s 414us/step - loss: 139.3933 -
val_loss: 159.6935
Epoch 45/50
721/721 [==============================] - 0s 444us/step - loss: 138.8230 -
val_loss: 161.4545
Epoch 46/50
721/721 [==============================] - 0s 668us/step - loss: 137.7073 -
val_loss: 159.9421
Epoch 47/50
721/721 [==============================] - 0s 388us/step - loss: 137.0972 -
val_loss: 158.8505
Epoch 48/50
721/721 [==============================] - 0s 381us/step - loss: 136.7723 -
val_loss: 161.7408
Epoch 49/50
721/721 [==============================] - 0s 395us/step - loss: 135.4033 -
val_loss: 159.8462
Epoch 50/50
721/721 [==============================] - 0s 393us/step - loss: 134.7356 -
val_loss: 160.0924
```

[41]: <keras.callbacks.History at 0x7f13ac342198>

6 Evaluate the model on the test data

[42]:
```
loss_val = model.evaluate(X_test, y_test)
y_pred = model.predict(X_test)
loss_val
```

```
309/309 [==============================] - 0s 72us/step
```

[42]: 87916561.94174758

6.0.1 compute the mean squared error between the predicted concrete strength and the actual concrete strength.

Let's import the mean_squared_error function from Scikit-learn.

[43]:
```
from sklearn.metrics import mean_squared_error
```

[44]:

```
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)
```

```
87916561.96664037 0.0
```

7 Build a Neural Network

[45]:

```
total_mean_squared_errors = 50
epochs = 50
mean_squared_errors = []
for i in range(0, total_mean_squared_errors):
    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=epochs, 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)

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_errors) + " mean squared errors with normalized data.
 ↪ Total number of epochs for each training is: " +str(epochs) + "\n")
print("Mean: "+str(mean))
print("Standard Deviation: "+str(standard_deviation))
```

```
MSE 1: 85.4928356034856
MSE 2: 71.79738034245266
MSE 3: 47.18585397664783
MSE 4: 42.18026948206633
MSE 5: 38.8063833242867
MSE 6: 37.007631888281566
MSE 7: 37.69336732537229
MSE 8: 27.486753192148548
MSE 9: 30.20358455374017
MSE 10: 28.857695866557002
MSE 11: 26.703263094510074
MSE 12: 20.743106379092318
MSE 13: 27.452345721544184
MSE 14: 28.127326261650012
```

```
MSE 15: 26.333046169342733
MSE 16: 18.16792730065997
MSE 17: 22.48954719864435
MSE 18: 23.21905312337536
MSE 19: 22.00694696185658
MSE 20: 25.15413333682952
MSE 21: 20.462140503053142
MSE 22: 21.63909628784772
MSE 23: 19.186820829570486
MSE 24: 21.396980612794945
MSE 25: 22.124526576316857
MSE 26: 23.00177013063894
MSE 27: 18.60725961765425
MSE 28: 20.022344194183844
MSE 29: 23.200144005439043
MSE 30: 20.07623579895612
MSE 31: 16.974975104470854
MSE 32: 18.51956346505668
MSE 33: 18.687298154367983
MSE 34: 19.014024969057743
MSE 35: 21.57144194667779
MSE 36: 24.872519965310698
MSE 37: 16.835172949485408
MSE 38: 20.703261464930662
MSE 39: 19.298146806488532
MSE 40: 17.368115187462866
MSE 41: 21.199221817806702
MSE 42: 15.572148233555668
MSE 43: 19.184246868763154
MSE 44: 19.45430984311891
MSE 45: 20.275742712915907
MSE 46: 18.583385307040416
MSE 47: 18.769526225463473
MSE 48: 17.802092913285044
MSE 49: 18.47012780939491
MSE 50: 18.579622299540006


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

Mean: 25.571213937870226
Standard Deviation: 12.828017348377932
```

[ ]:

```

```