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.

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

[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
```

[6]:
```
concrete_data.shape #data points
```

[6]:
```
(1030, 9)
```

[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
```

[7]:
```
concrete_data.isnull().sum() # CLEAN THE DATA
```

[7]:
```
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

[8]:
```
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
```

[9]:
```
predictors.head()
```

[9]:
```
   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
```

[10]:
```
target.head()
```

[10]:
```
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.

[11]:
```
predictors_norm = (predictors - predictors.mean()) / predictors.std()
predictors_norm.head()
```

[11]:
```
     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

[13]:
```python
n_cols = predictors_norm.shape[1] # number of predictors
```

3 Import the Keras library

[14]:
```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)])
```

[15]:
```
from keras.models import Sequential
from keras.layers import Dense
```

4 Build a Neural Network

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.

[17]:
```
# 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(1))

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

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

[18]:
```
from sklearn.model_selection import train_test_split
```

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

[20]:
```
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

[21]:
```
# 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.

[23]:
```
# 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 [==============================] - 0s 412us/step - loss: 158.8141 -
```

```
val_loss: 106.1055
Epoch 2/50
721/721 [==============================] - 0s 445us/step - loss: 157.2554 -
val_loss: 106.7075
Epoch 3/50
721/721 [==============================] - 0s 416us/step - loss: 155.6376 -
val_loss: 107.5418
Epoch 4/50
721/721 [==============================] - 0s 357us/step - loss: 154.1690 -
val_loss: 107.9913
Epoch 5/50
721/721 [==============================] - 0s 368us/step - loss: 152.7373 -
val_loss: 108.5248
Epoch 6/50
721/721 [==============================] - 0s 397us/step - loss: 151.0639 -
val_loss: 109.8685
Epoch 7/50
721/721 [==============================] - 0s 437us/step - loss: 149.7963 -
val_loss: 110.5298
Epoch 8/50
721/721 [==============================] - 0s 392us/step - loss: 148.3151 -
val_loss: 110.5286
Epoch 9/50
721/721 [==============================] - 0s 409us/step - loss: 146.9782 -
val_loss: 111.2455
Epoch 10/50
721/721 [==============================] - 0s 361us/step - loss: 145.7472 -
val_loss: 112.8683
Epoch 11/50
721/721 [==============================] - 0s 415us/step - loss: 144.3118 -
val_loss: 113.2248
Epoch 12/50
721/721 [==============================] - 0s 383us/step - loss: 143.0453 -
val_loss: 114.0196
Epoch 13/50
721/721 [==============================] - 0s 442us/step - loss: 142.1584 -
val_loss: 115.1693
Epoch 14/50
721/721 [==============================] - 0s 381us/step - loss: 140.8834 -
val_loss: 116.3022
Epoch 15/50
721/721 [==============================] - 0s 390us/step - loss: 139.5412 -
val_loss: 116.5498
Epoch 16/50
721/721 [==============================] - 0s 383us/step - loss: 138.5614 -
val_loss: 117.9367
Epoch 17/50
721/721 [==============================] - 0s 388us/step - loss: 137.1343 -
```

```
val_loss: 118.5434
Epoch 18/50
721/721 [==============================] - 0s 385us/step - loss: 136.1445 -
val_loss: 120.3195
Epoch 19/50
721/721 [==============================] - 0s 360us/step - loss: 134.9155 -
val_loss: 121.5792
Epoch 20/50
721/721 [==============================] - 0s 365us/step - loss: 133.8227 -
val_loss: 121.2784
Epoch 21/50
721/721 [==============================] - 0s 359us/step - loss: 132.7879 -
val_loss: 122.6053
Epoch 22/50
721/721 [==============================] - 0s 472us/step - loss: 131.7246 -
val_loss: 123.6060
Epoch 23/50
721/721 [==============================] - 0s 410us/step - loss: 130.6438 -
val_loss: 124.5070
Epoch 24/50
721/721 [==============================] - 0s 522us/step - loss: 129.6374 -
val_loss: 125.9104
Epoch 25/50
721/721 [==============================] - 0s 447us/step - loss: 128.9126 -
val_loss: 127.3362
Epoch 26/50
721/721 [==============================] - 0s 389us/step - loss: 127.8902 -
val_loss: 128.3615
Epoch 27/50
721/721 [==============================] - 0s 438us/step - loss: 127.0193 -
val_loss: 128.7330
Epoch 28/50
721/721 [==============================] - 0s 415us/step - loss: 125.9038 -
val_loss: 129.0002
Epoch 29/50
721/721 [==============================] - 0s 410us/step - loss: 124.8321 -
val_loss: 129.0912
Epoch 30/50
721/721 [==============================] - 0s 417us/step - loss: 123.7985 -
val_loss: 131.1239
Epoch 31/50
721/721 [==============================] - 0s 390us/step - loss: 122.9441 -
val_loss: 132.4291
Epoch 32/50
721/721 [==============================] - 0s 442us/step - loss: 122.2538 -
val_loss: 133.5568
Epoch 33/50
721/721 [==============================] - 0s 390us/step - loss: 121.3691 -
```

```
val_loss: 132.2868
Epoch 34/50
721/721 [==============================] - 0s 438us/step - loss: 120.4920 -
val_loss: 134.6746
Epoch 35/50
721/721 [==============================] - 0s 473us/step - loss: 119.8096 -
val_loss: 135.7861
Epoch 36/50
721/721 [==============================] - 0s 414us/step - loss: 118.9906 -
val_loss: 136.8689
Epoch 37/50
721/721 [==============================] - 0s 448us/step - loss: 118.0742 -
val_loss: 136.6626
Epoch 38/50
721/721 [==============================] - 0s 382us/step - loss: 117.2895 -
val_loss: 137.0989
Epoch 39/50
721/721 [==============================] - 0s 389us/step - loss: 116.6981 -
val_loss: 136.3788
Epoch 40/50
721/721 [==============================] - 0s 421us/step - loss: 115.8347 -
val_loss: 138.1166
Epoch 41/50
721/721 [==============================] - 0s 688us/step - loss: 115.2098 -
val_loss: 138.9060
Epoch 42/50
721/721 [==============================] - 0s 415us/step - loss: 114.5285 -
val_loss: 139.1216
Epoch 43/50
721/721 [==============================] - 0s 416us/step - loss: 113.5577 -
val_loss: 140.4369
Epoch 44/50
721/721 [==============================] - 0s 637us/step - loss: 112.9891 -
val_loss: 140.6825
Epoch 45/50
721/721 [==============================] - 0s 411us/step - loss: 112.3483 -
val_loss: 141.5391
Epoch 46/50
721/721 [==============================] - 0s 496us/step - loss: 111.5387 -
val_loss: 142.1855
Epoch 47/50
721/721 [==============================] - 0s 419us/step - loss: 111.2179 -
val_loss: 140.9655
Epoch 48/50
721/721 [==============================] - 0s 416us/step - loss: 110.5176 -
val_loss: 139.9818
Epoch 49/50
721/721 [==============================] - 0s 410us/step - loss: 110.3294 -
```

```
val_loss: 143.7165
Epoch 50/50
721/721 [==============================] - 0s 386us/step - loss: 109.2572 -
val_loss: 142.5937
```

[23]: <keras.callbacks.History at 0x7fee15c44390>

6 Evaluate the model on the test data

[24]:
```
loss_val = model.evaluate(X_test, y_test)
y_pred = model.predict(X_test)
loss_val
```

```
309/309 [==============================] - 0s 129us/step
```

[24]: 41407465.32038835

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.

[25]:
```
from sklearn.metrics import mean_squared_error
```

[26]:
```
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)
```

```
41407464.414271615 0.0
```

7 Create a list of 50 mean squared errors and report mean and the standard deviation of the mean squared errors.

[27]:
```
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, 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("Below is the mean and standard deviation of "␣
↪+str(total_mean_squared_errors) + " mean squared errors without 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: 2485.8806255056634
MSE 2: 364.48710365665767
MSE 3: 244.60458502414542
MSE 4: 270.6536970910131
MSE 5: 253.46854047559225
MSE 6: 215.75344305439674
MSE 7: 257.0338008349767
MSE 8: 184.53906190742566
MSE 9: 210.34155293189977
MSE 10: 183.46824670686692
MSE 11: 169.52499508163305
MSE 12: 164.02616916891054
MSE 13: 146.459690501389
MSE 14: 142.36362751093498
MSE 15: 100.41088844965962
MSE 16: 54.366279293418316
MSE 17: 55.93005685898864
MSE 18: 51.60272004457739
MSE 19: 41.15331311519092
MSE 20: 46.940897105195376
MSE 21: 43.17192497376871
MSE 22: 42.603018134157246
MSE 23: 39.24455702574893
MSE 24: 40.623416715455285
MSE 25: 45.03258888466844
MSE 26: 45.45151618537779
MSE 27: 45.71576132666332
MSE 28: 40.48105153451074
MSE 29: 46.40764478418048
MSE 30: 43.98078355511415
MSE 31: 45.48971606998382
MSE 32: 37.09484625325619
MSE 33: 41.74664067141832
MSE 34: 40.72372613061207
MSE 35: 37.425077888957894
```

```
MSE 36: 45.64947622725107
MSE 37: 46.295517183816166
MSE 38: 42.44215817744678
MSE 39: 38.2584443509
MSE 40: 39.126816610688145
MSE 41: 43.44314438162498
MSE 42: 37.77803346948716
MSE 43: 43.21549740886997
MSE 44: 41.35343080045336
MSE 45: 51.65417371830122
MSE 46: 43.734879070886905
MSE 47: 41.88216848280823
MSE 48: 43.21275382674628
MSE 49: 39.55350061064785
MSE 50: 44.282121417591874
Below is the mean and standard deviation of 50 mean squared errors without
normalized data. Total number of epochs for each training is: 50

Mean: 138.40167421886906
Standard Deviation: 345.0033454170249
```

8 THANK YOU