Linear Regression

In this notebook, we will learn how to apply Linear regression for predicting the compressive strength of concrete.

The attached dataset is taken from the UC Irvine Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength).

To run this code, you will need the following python packages:

- numpy
- pandas
- scikit-learn

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

In [3]:
```
# First, we load the dataset using pandas
df = pd.read_excel("Concrete_Data.xlsx")
```

In [4]:
```
# next, we will split the dataframe into a training and testing splits with a 70% / 30% ratio
from sklearn.model_selection import train_test_split

df_train, df_test = train_test_split(df, test_size=0.3, random_state=42) # Random is fixed for re
producability
```

In [5]:
```
# Now lets display a few rows from the training data
df_train
```

Out[5]:

	Cement (component 1)(kg in a m^3 mixture)	Blast Furnace Slag (component 2) (kg in a m^3 mixture)	Fly Ash (component 3)(kg in a m^3 mixture)	Water (component 4)(kg in a m^3 mixture)	Superplasticizer (component 5) (kg in a m^3 mixture)	Coarse Aggregate (component 6) (kg in a m^3 mixture)	Fine Aggregate (component 7) (kg in a m^3 mixture)	Age (day)	Concrete compressive strength(MPa, megapascals)
196	194.68	0.0	100.52	165.62	7.48	1006.4	905.90	28	25.724350
631	325.00	0.0	0.00	184.00	0.00	1063.0	783.00	7	17.540269
81	318.80	212.5	0.00	155.70	14.30	852.1	880.40	3	25.200348
526	359.00	19.0	141.00	154.00	10.91	942.0	801.00	3	23.639177
830	162.00	190.0	148.00	179.00	19.00	838.0	741.00	28	33.756745
...	...	...	...	...	...	...	...	...	...
87	286.30	200.9	0.00	144.70	11.20	1004.6	803.70	3	24.400556
330	246.83	0.0	125.08	143.30	11.99	1086.8	800.89	14	42.216615
466	190.34	0.0	125.18	166.61	9.88	1079.0	798.90	100	33.563692
121	475.00	118.8	0.00	181.10	8.90	852.1	781.50	28	68.299493
860	314.00	0.0	113.00	170.00	10.00	925.0	783.00	28	38.458971

721 rows × 9 columns

In [6]:
```
# Then lets view some statistics
df_train.describe()
```

Out[6]:

	Cement (component 1)(kg in a m^3 mixture)	Blast Furnace Slag (component 2)(kg in a m^3 mixture)	Fly Ash (component 3)(kg in a m^3 mixture)	Water (component 4)(kg in a m^3 mixture)	Superplasticizer (component 5)(kg in a m^3 mixture)	Coarse Aggregate (component 6)(kg in a m^3 mixture)	Fine Aggregate (component 7)(kg in a m^3 mixture)	Age (day)	Concrete compressive strength(MPa, megapascals)
count	721.000000	721.000000	721.000000	721.000000	721.000000	721.000000	721.000000	721.000000	721.000000
mean	284.409681	74.971886	52.006588	181.805576	6.125337	973.798128	771.636297	46.049931	36.152573
std	108.361334	87.717335	63.707358	21.159956	6.046367	78.509208	80.125492	61.650743	16.803402
min	102.000000	0.000000	0.000000	121.750000	0.000000	801.000000	594.000000	1.000000	2.331808
25%	192.000000	0.000000	0.000000	165.620000	0.000000	932.000000	724.300000	14.000000	23.890343
50%	277.000000	22.000000	0.000000	185.700000	6.000000	968.000000	778.450000	28.000000	35.076402
75%	362.600000	145.000000	117.540000	192.000000	10.100000	1040.000000	821.000000	56.000000	46.247292
max	540.000000	359.400000	195.000000	247.000000	32.200000	1145.000000	992.600000	365.000000	82.599225

In [7]:
```
# Now we will extract the models input and targets from both the training and testing dataframes
def extract_Xy(df):
    df_numpy = df.to_numpy()
    return df_numpy[:, :-1], df_numpy[:, -1]

X_train, y_train = extract_Xy(df_train)
X_test, y_test = extract_Xy(df_test)
```

Linear Regression via Scikit-Learn

In [8]:
```
# Then we test the linear regression using Scikit-learn's implementation
from sklearn.linear_model import LinearRegression

model = LinearRegression().fit(X_train, y_train)
```

In [9]:
```
# Using scikit-learn's MSE function, we can compute the training and testing error for our model
from sklearn.metrics import mean_squared_error

y_train_predict = model.predict(X_train)
training_error = mean_squared_error(y_train, y_train_predict)
print(f"Training Error: {training_error} (RMS: {training_error**0.5})")
y_test_predict = model.predict(X_test)
testing_error = mean_squared_error(y_test, y_test_predict)
print(f"Testing Error: {testing_error} (RMS: {testing_error**0.5})")

#Note: We also display the Root Mean Square error (RMS) since it is more intuitive to compare wit
h the dataset statistics (diplayed using df_train.describe())
```

```
Training Error: 107.25842311011506 (RMS: 10.356564252208116)
Testing Error: 109.75614063734936 (RMS: 10.47645649240951)
```

In [10]:
```
%%timeit
LinearRegression().fit(X_train, y_train)
# Here we are measuring the training time to compare with our implementation below
```

```
311 µs ± 16.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Linear Regression from Scratch

In [11]:
```
def our_mean_square_error(true, predicted):
    #TODO: implement this function to match Scikit-learn's mean_square_error
    #Note: both true & predicted will be float numpy arrays
    return ((true - predicted)**2).mean()
```

In [12]:
```
print(f"{our_mean_square_error( np.array([  1, 0]), np.array([1,   0]) ) = }") # Should be 0
print(f"{our_mean_square_error( np.array([  0, 1]), np.array([1,   0]) ) = }") # Should be 1
print(f"{our_mean_square_error( np.array([0.5, 0]), np.array([1, 0.5]) ) = }") # Should be 0.25
```

```
our_mean_square_error( np.array([  1, 0]), np.array([1,   0]) ) = 0.0
our_mean_square_error( np.array([  0, 1]), np.array([1,   0]) ) = 1.0
our_mean_square_error( np.array([0.5, 0]), np.array([1, 0.5]) ) = 0.25
```

In [13]:
```
class OurLinearRegression:
    def _prepare_inputs(self, X):
        # Here, we add a new input with value 1 to each example. It will be multipled by the bias
        ones = np.ones((X.shape[0], 1), dtype=X.dtype)
        return np.concatenate((ones, X), axis=1)

    def fit(self, X, y):
        X = self._prepare_inputs(X) # First, we prepare the inputs
        #TODO: compute and store the model weights into self.w
        # Note: you can use numpy function and do not use "numpy.linalg.lstsq" or "numpy.linalg.p
inv"
        # To compute a square matrix's inverse, you can use "numpy.linalg.inv".
        # A more stable option to compute "numpy.linalg.inv(A) @ b" is using "numpy.linalg.solve
(A, b)"

        trans = X.transpose()
        self.w = np.linalg.solve(np.matmul(trans, X), np.matmul(trans, y))

        # Return self to match the behavior of Scikit-Learn's LinearRegression fit()
        return self

    def predict(self, X):
        X = self._prepare_inputs(X) # First, we prepare the inputs
        #TODO: Compute and return the predictions given X
        return np.matmul( X , self.w)
```

In [14]:
```
# Now, you can train your model
our_model = OurLinearRegression().fit(X_train, y_train)
```

In [15]:
```
# Using your MSE function, you can compute the training and testing error for our model
y_train_predict = our_model.predict(X_train)
training_error = our_mean_square_error(y_train, y_train_predict)
print(f"Training Error: {training_error} (RMS: {training_error**0.5})")
y_test_predict = our_model.predict(X_test)
testing_error = our_mean_square_error(y_test, y_test_predict)
print(f"Testing Error: {testing_error} (RMS: {testing_error**0.5})")
```

```
Training Error: 107.25842311011506 (RMS: 10.356564252208116)
Testing Error: 109.75614063734861 (RMS: 10.476456492409474)
```

In [16]:
```
%%timeit
OurLinearRegression().fit(X_train, y_train)
# Now, you can compare the time of our implementation with Scikit-Learn's. What is your conclusio
n?
```

```
30.2 µs ± 1.5 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

In [17]:
```
#TODO: Write your conclusion about your implementation's performance and training time
```

The traning and testing errors are almost identical to the sklearn implementation with a great reduction in time than the sklearn implemntation