Machine Learning

project III

By 費蓋德 0516251

Environment

- Python3 through jupyterlab.
- Imported the following libraries (and use most of them):

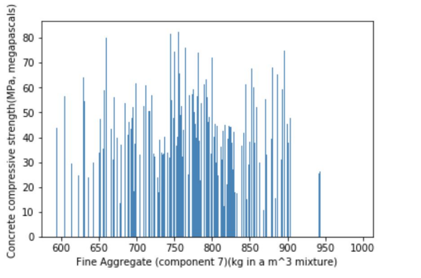
```python
import pandas as pd
import numpy as np
import sklearn
from sklearn import datasets
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt
import seaborn as sns
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
from math import sqrt
import statsmodels.formula.api as smf
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
```

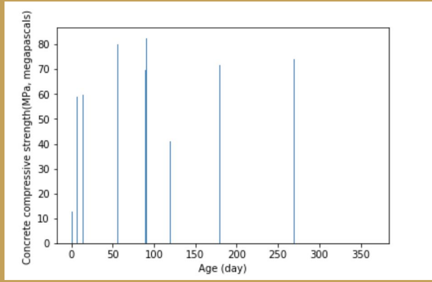
Visualization

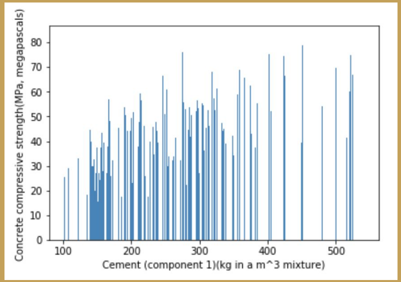
Each feature plotted separately with the target

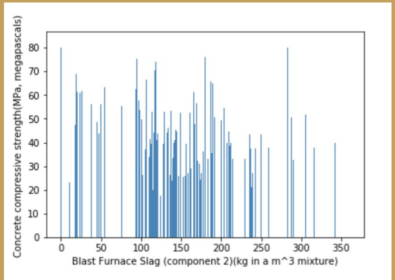
Code for plotting the graphs

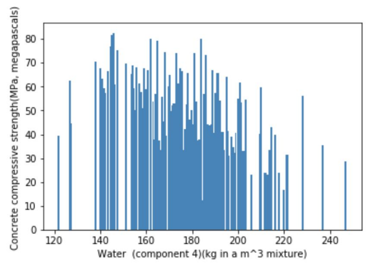
```
# plot every feature with the output
for column in df_features:
    fig, ax = plt.subplots()
    plt.bar(df_features[column], df['Concrete compressive strength(MPa, megapascals) '])
    ax.set_xlabel(column)
    ax.set_ylabel('Concrete compressive strength(MPa, megapascals) ')
    plt.show()
```

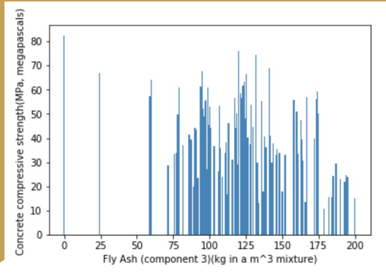


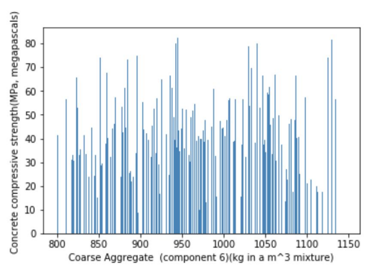


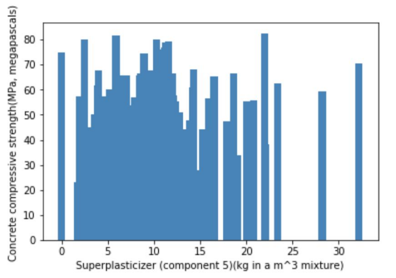










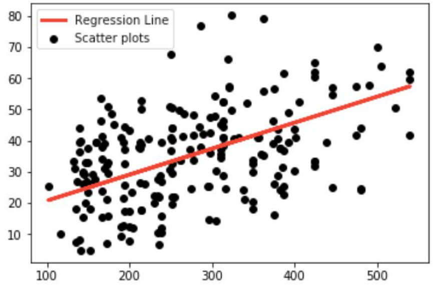


Problem I

Code, Graph, accuracy, Weight and Bias

```
# UNIVARIATE LR USING THE MOST RELEVANT FEATURE (cement) --> from sklearn library
array = np.array(df)
X = array[:,0].reshape(-1, 1)
Y = array[:,8]

# weight and bias
weight = ((np.mean(X)*np.mean(Y))-np.mean(X*Y))/((np.mean(X)*np.mean(X))-np.mean(X*X))
print("Weight :", weight)
bias = np.mean(Y)-np.mean(X)*weight
print("Bias: ", bias)

# Linear regression
train_X, test_X, train_y, test_y = train_test_split(X, Y, test_size=0.20)
reg = LinearRegression().fit(train_X, train_y)
pred_y = reg.predict(test_X)

# accuracy
acc = reg.score(test_X, test_y)
print("accuracy: ", acc)

# Plot the graph
plt.scatter(test_X, test_y,  color='black', label = 'Scatter plots')
plt.plot(test_X, pred_y, color='red', linewidth=3, label='Regression Line')
plt.legend()
plt.show()
```

```
('Weight :', -1.6671187614337197e-16)
('Bias: ', 35.817961165048594)
('accuracy: ', 0.17424388692370807)
```



I used correlation matrix to select the most relevant features and it turns out that it's ***cement*** so we use it for univariate LR.

I used the sklearn linear regression library

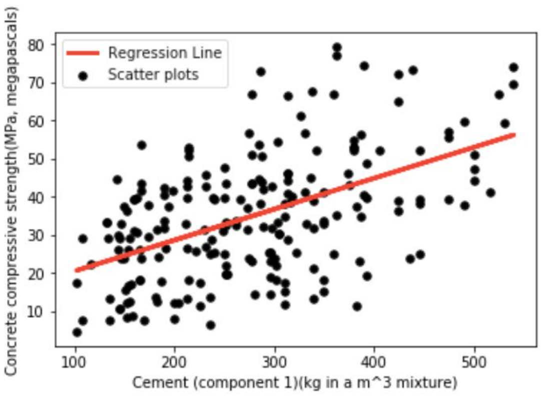
Problem 2

Code, Graph, accuracy, Weight and Bias

```python
# by my own gd
def estimate_coef(x, y):
    # number of observations/points
    n = np.size(x)

    # mean of x and y vector
    m_x, m_y = np.mean(x), np.mean(y)

    # calculating cross-deviation and deviation about x
    SS_xy = np.sum(y*x) - n*m_y*m_x
    SS_xx = np.sum(x*x) - n*m_x*m_x

    # calculating regression coefficients
    b_1 = SS_xy / SS_xx
    b_0 = m_y - b_1*m_x

    return(b_0, b_1)

def plot_regression_line(x, y, b):
    # plotting the actual points as scatter plot
    plt.scatter(x, y, color = "black",
               marker = "o", s = 30, label = 'Scatter plots')

    # predicted response vector
    y_pred = b[0] + b[1]*x

    # plotting the regression line
    plt.plot(x, y_pred, color = "red", linewidth=3, label='Regression Line')
    plt.xlabel('Cement (component 1)(kg in a m^3 mixture)')
    plt.ylabel('Concrete compressive strength(MPa, megapascals)')
    plt.legend()
    plt.show()
def main():
    # weight and bias
    weight = ((np.mean(X)*np.mean(Y))-np.mean(X*Y))/((np.mean(X)*np.mean(X))-np.mean(X*X))
    print("Weight :", weight)
    bias = np.mean(Y)-np.mean(X)*weight
    print("Bias: ", bias)

    train_X, test_X, train_y, test_y = train_test_split(X, Y, test_size=0.20)
    test_X = test_X.flatten()
    test_y = test_y.flatten()
    train_X = train_X.flatten()
    train_y = train_y.flatten()
    # estimating coefficients
    b = estimate_coef(test_X, test_y)
    # plotting regression line
    plot_regression_line(test_X, test_y, b)

    # claculating accuracy
    ss_t = 0
    ss_r = 0
    for i in range(m):
        y_pred = b[0] + b[1] * X[i]
        ss_t += (Y[i] - np.mean(Y)) ** 2
        ss_r += (Y[i] - y_pred) ** 2
    r2 = 1 - (ss_r/ss_t)
    print("Accuracy: ", r2)
```

Better accuracy than the sklearn model

```
('Weight :', -1.6671187614337197e-16)
('Bias: ', 35.817961165048594)
```




```
('Accuracy: ', array([0.24617704]))
```

Comparison between problem 1 and 2

Similarities

- Both are linear regression models
- They both have low accuracy
- They have same weight
- And also same bias

Differences

- One is from a preexisting library, the other is from my own GD
- Problem 2 has greater accuracy

```python
# multivariate linear regression
# inputs
cement = df['Cement (component 1)(kg in a m^3 mixture)'].values
blast = df['Blast Furnace Slag (component 2)(kg in a m^3 mixture)'].values
ash = df['Fly Ash (component 3)(kg in a m^3 mixture)'].values
water = df['Water  (component 4)(kg in a m^3 mixture)'].values
plasticizer = df['Superplasticizer (component 5)(kg in a m^3 mixture)'].values
coarse = df['Coarse Aggregate  (component 6)(kg in a m^3 mixture)'].values
fine = df['Fine Aggregate (component 7)(kg in a m^3 mixture)'].values
age = df['Age (day)'].values

# output
res = df['Concrete compressive strength(MPa, megapascals) '].values


m = len(cement)
x0 = np.ones(m)
X = np.array([x0, cement, blast, ash, water, plasticizer, coarse, fine, age]).T

# Initial Coefficients
B = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0])
Y = np.array(res)

train_X, test_X, train_y, test_y = train_test_split(X, Y, test_size=0.20)

alpha = 0.0000001

def cost_function(X, Y, B):
    m = len(Y)
    J = np.sum((X.dot(B) - Y) ** 2)/(2 * m)
    return J

inital_cost = cost_function(train_X, train_y, B)
print(inital_cost)


def gradient_descent(X, Y, B, alpha, iterations):
    cost_history = [0] * iterations
    m = len(Y)

    for iteration in range(iterations-1):
        # Hypothesis Values
        h = X.dot(B)
        # Difference b/w Hypothesis and Actual Y
        loss = h - Y
        # Gradient Calculation
        gradient = X.T.dot(loss) / m
        # Changing Values of B using Gradient
        B = B - (alpha * gradient)
        # New Cost Value
        cost = cost_function(X, Y, B)
        cost_history[iteration] = cost

    return B, cost_history

# 100000 Iterations
newB, cost_history = gradient_descent(train_X, train_y, B, alpha, 100000)

# Model Evaluation - RMSE
def rmse(Y, Y_pred):
    rmse = np.sqrt(sum((Y - Y_pred) ** 2) / len(Y))
    return rmse

# Model Evaluation - R2 Score
def r2_score(Y, Y_pred):
    mean_y = np.mean(Y)
    ss_tot = sum((Y - mean_y) ** 2)
    ss_res = sum((Y - Y_pred) ** 2)
    r2 = 1 - (ss_res / ss_tot)
    return r2

Y_pred = test_X.dot(newB)

print("The MSE of Multivariate LR: ", rmse(test_y, Y_pred))
print("The R^2 of Multivariate LR: ", r2_score(test_y, Y_pred))
```

```
778.5600507888349
('The MSE of Multivariate LR: ', 9.49166609931123)
('The R^2 of Multivariate LR: ', 0.6634534778246967)
```

Problem 3

Code, MSE, R-squared and performance comparison between update methods

As shown in the picture

R_Squared = Accuracy = 0.66
MSE = 9.50

Performance comparison between 2 different update methods

The greater our learning rate is, the less iterations we should have and the less performant our algo is. For example, when we have 100 iterations and a learning rate of 0.00001, our performance is as follows:

```
('The MSE of Multivariate LR: ', 13.337506637017732)
('The R^2 of Multivariate LR: ', 0.3214555546369978)
```

Thus, we should find a balance for the learning rate and the number of iterations to prevent underfitting or overfitting.

Problem 4

Code, MSE, R-squared

```
array = np.array(df)
X = np.array(df_features)
Y = array[:,8]
train_X, test_X, train_y, test_y = train_test_split(X, Y, test_size=0.20)

model = Pipeline([('poly', PolynomialFeatures(degree=3)),
                  ('linear', LinearRegression(fit_intercept=False))])

model = model.fit(train_X, train_y)
y_pred = model.predict(test_X)
yp = np.array(y_pred)
y  = np.array(test_y)
r = r2_score(y, yp)
mse = mean_squared_error(y, yp)
print("mse for polynomial reg: ", mse)
print("r^2 for polynomial reg: ", r)
```

```
('mse for polynomial reg: ', 41.66177668646611)
('r^2 for polynomial reg: ', 0.8754027907813978)
```

Multivariate polynomial regression can reach an **accuracy > 87** % as shown in the screenshot

Part 9: Answer to the questions

Overfitting:

It happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data. That's why it's important (in our case) to know how to tune the learning rate and number of iterations.

Benefit of SGD:

On large datasets, SGD can **converge faster** than batch training because it **performs updates more frequently**. We can get away with this because the data often contains redundant information, so the gradient can be **reasonably approximated without using the full dataset.**

Why the different initial value to GD model may cause different result ?

Because the fitting line will converge regarding the initial value. The better the initial value, the best the line will fit the data. A bad initial value will converge to a local minimum which is not necessarily the global minimum. So it is important that the initial value is good (not necessarily perfect).

What is the bad learning rate? What problem will happen if we use it?

A bad learning rate is one which is **too large or too small**.

- If it's too large it will overshoot the minima preventing the hypothesis from converging.
- If it's too small, it will take too long to converge and will require lots of iterations

Problem encountered and how they were solved

At some point the learning rate was too small for the number of iterations I set, This caused overflow. By trial and error, I found the ideal number of iterations for the ideal learning rate.

費蓋德 0516251