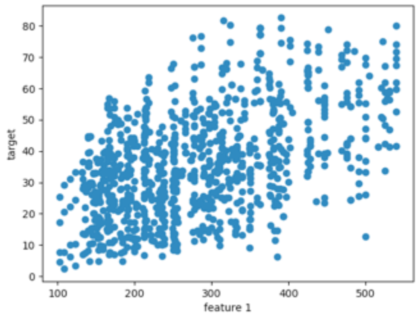
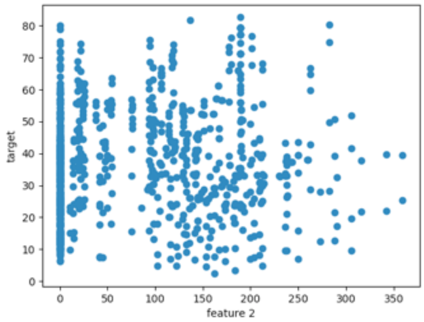
Introduction to Machine Learning Program Assignment #3

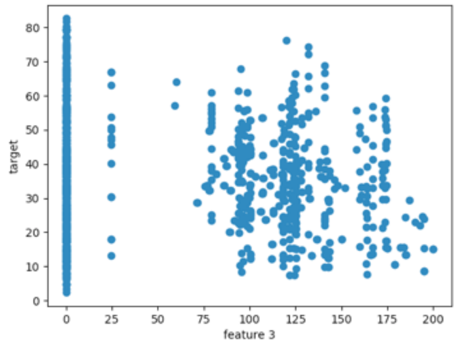
Members:
0510002 袁鈺勛
0510020 方鈺豪
0510022 劉孟寰
0510023 李佳任
0086043 陳以嫵

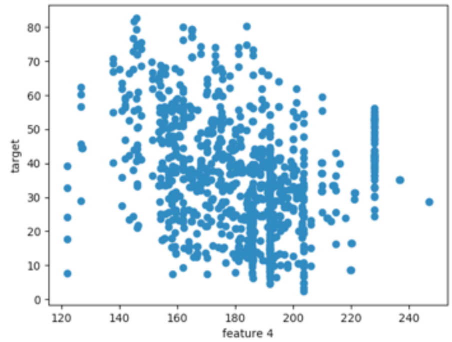
(1)
What environments the members are using?
Ans:
We use python to implement Machine Learning.
We use three packages: Pandas , Matplotlib and Sklearn.
Pandas package is convenient for processing the CSV file.
Matplotlib package is useful for visualizing the outcome.
Sklearn has many useful functions such as splitting test and train datas and computing r2_score and mean_squared_error, also , we complete problem 1 by built-in function provided by it.

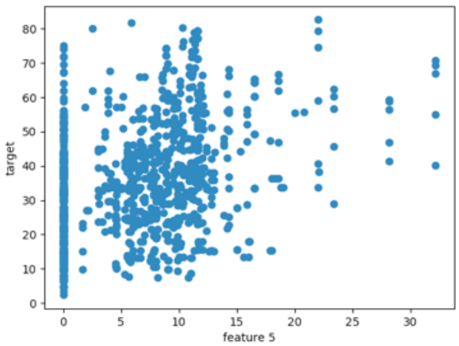
(2)
Visualizaton of all the feature with the target
Ans:

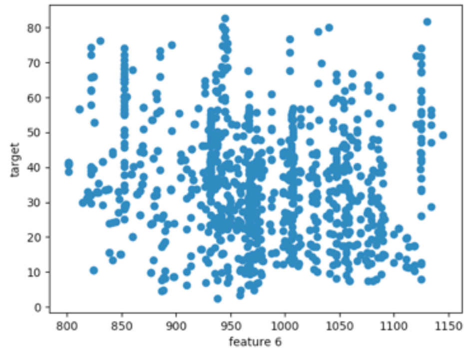


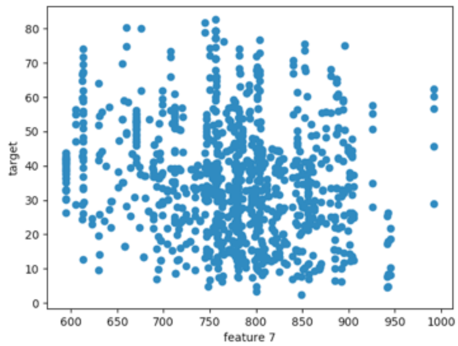


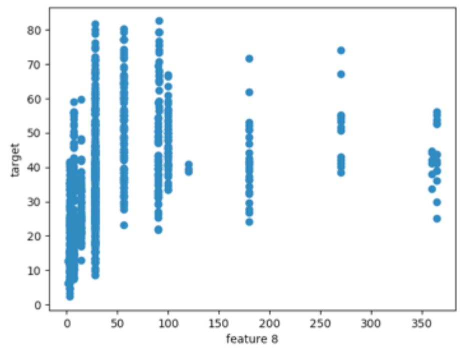












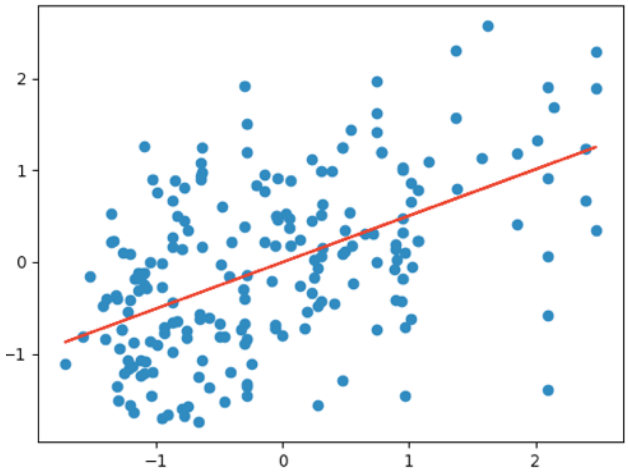
feature 1-8 refer to Cement , Blast Furnace Slag , Fly Ash , Water , Superplasticizer , Coarse Aggregate , Fine Aggregate and Age. The target is Concrete compressive strength.
First we read the csv file , reshape the target data to one column , and then use iloc function to select specific column(ie. specific feature data) to draw scatter plot with respect to the target data.
We can see from the graph that cement(first feature) is the most relative feature, so we use it in the following questions.

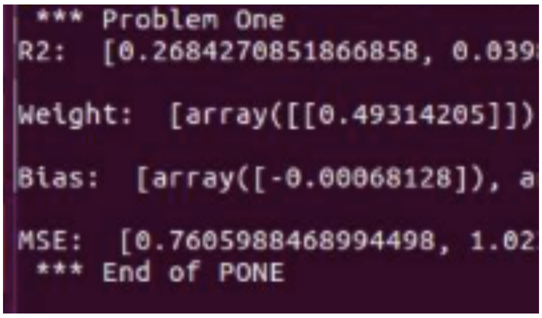
(3)
The code, graph, r2_score, weight and bias for problem 1
Ans:

```
def problemOne(concrete, regressand):
        print(" *** Problem One")

        scaler = StandardScaler()
        regressand = scaler.fit_transform(regressand)
        r2 = []
        weight = []
        bias = []
        mse = []
        for i in range(8):
                regressor = concrete.iloc[:, i].values.reshape(-1, 1)
                regressor = regressor.astype(float)
```

```
        regressor = scaler.fit_transform(regressor)
        plt.scatter(regressor, regressand)
        plt.show()

        x_train, x_test, y_train, y_test = train_test_split(regressor, regressand, test_size = 0.2)
        lr = LinearRegression()
        lr.fit(x_train, y_train)
        pred = lr.predict(x_test)

        r2.append(r2_score(y_test, pred))
        weight.append(lr.coef_)
        bias.append(lr.intercept_)
        mse.append(mean_squared_error(y_test, pred))

        plt.scatter(x_test, y_test)
        plt.plot(x_test, pred, color = 'red')
        plt.show()

    print("R2: ", r2, "\n")
    print("Weight: ", weight, "\n")
    print("Bias: ", bias, "\n")
    print("MSE: ", mse)
    print(" *** End of PONE\n\n")
```

For each feature, because the data ranges differ a lot, first we use StandardScaler in Sklearn library to standardlize our regressed and regressor (transform data such that its distribution will have a mean value 0 and standard deviation of 1) so that it is more efficient to do linear regression. Then we split the data and use LinearRegression() model to train, get the weight and bias, and predict the test data. We use built-in function to calculate the r2_score and mse for predict result and actual test data.



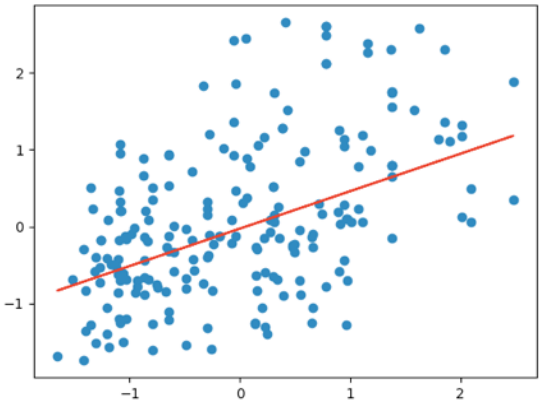


```
 *** Problem One
R2:  [0.2684270851866858, 0.039
Weight:  [array([[0.49314205]])
Bias:  [array([-0.00068128]), a
MSE:  [0.7605988468994498, 1.02
 *** End of PONE
```

Because we do single variable linear regression for all the features , so we have a list of r2_score, weight , bias and mse for each of them, here we just show the result and graph of the first feature.The scatter plot contains testing datas spots.

(4)
The code, graph, r2_score, weight and bias for problem 2
Ans:
```
def problemTwo(concrete, regressand):
	print(" *** Problem Two")

	r2 = []
	weight = []
	bias = []
	mse = []
	scaler = StandardScaler()
	regressand = scaler.fit_transform(regressand)

	for i in range(8):
		regressor = concrete.iloc[:, i]
		regressor = regressor.astype(float)
		regressor = np.c_[np.ones(len(regressand)), regressor]
		regressor = scaler.fit_transform(regressor)
		for j in range(len(regressor)):
			regressor[j][0] = 1.0

		x_train, x_test, y_train, y_test = train_test_split(regressor, regressand, test_size = 0.2)

		init_theta = np.array([np.ones(2)])
		theta = gradientDescent(x_train, y_train, init_theta, 1.0e-3, 1.0e-8)
		pred = np.matmul(x_test, theta.T)
		pred_train = np.matmul(x_train, theta.T)

		r2.append(r2_score(y_test, pred))
		weight.append(theta[0][0])
		bias.append(theta[0][1])
		mse.append(mean_squared_error(y_train, pred_train))
		plotLine(x_test[:, 1], y_test, pred)

	print("R2: ", r2, "\n")
	print("Weight: ", weight, "\n")
	print("Bias: ", bias, "\n")
	print("MSE: ", mse)

	print(" *** End of PTWO\n\n”)

def gradientDescent(x, y, theta, alpha, precision):
	startTime = time.localtime(time.time())
	currentTime = time.localtime(time.time())
	prev_size = 1
	while(not(currentTime.tm_min >= startTime.tm_min+2 and currentTime.tm_sec >= startTime.tm_sec) and
prev_size > precision):
		prev_theta = theta
		theta = theta - alpha*np.sum((np.matmul(x, theta.T) - y)*x, axis = 0)
		prev_size = distance.euclidean(prev_theta[0], theta[0])
		currentTime = time.localtime(time.time())

	return theta
```

We build our own gradienDescent model.According to our cost function for linear regression: $f(w)$=(wx[j]−y[j])^2/2 for j=1 to n, our gradient descent is d$f(w)$=(wx[j]−y[j])*x[j] for j=1 to n. For matrix way, we update our theta according to formula: theta = theta - alpha*np.sum((np.matmul(x, theta.T) - y)*x, axis = 0) every iterateion until the difference between two theta is smaller than precision(means it comes to a minimum value) or time expires.Then we get our theta value for linear regression(theta is a matrix including weight and bias). We use built-in function to calculate the r2_score and mse for our predict result and actual test data.




```
 *** Problem Two
R2:  [0.2796577027224286, -0.0
210246655, 0.11445945563278714
47095578207651]

Weight:  [0.4824241450686858,
297362902, 0.3728251397302534,
254178404539]

Bias:  [-0.019475044602813723,
015826397385235766, -0.0037599
826957, -0.005906596216158953]

MSE:  [0.7454907202132404, 0.9
70156, 0.8537262334387195, 0.9
72197]
 *** End of PTWO
```

Because we do single variable linear regression for all the features , so we have a list of r2_score, weight , bias and mse for each of them, here we just show the result and graph of the first feature.The scatter plot contains testing datas spots.

(5)
Compare problem1 and problem 2, show what you got
Ans:
r2_score:0.269 and 0.280
weight:0.493 and 0.482
bias:-0.001 and -0.019
mse:0.761 and 0.745
r2_score of p2 is a little bit higher than p1, others are very similar.Maybe it is because we choose an appropriate theta according to observation and alpha and a right way to judge whether the iteration should stop.

(6)
The code, MSE, and the r2_score for problem 3

Ans:

```
def problemThree(concrete, regressand):
    print(" *** Problem Three")

    scaler = StandardScaler()
    regressand = scaler.fit_transform(regressand)
    regressor = concrete.iloc[:, [0, 1, 2, 3, 4, 5, 6, 7]]
    regressor = regressor.astype(float)
    regressor = np.c_[np.ones(len(regressand)), regressor]
    regressor = scaler.fit_transform(regressor)
    for i in range(len(regressor)):
        regressor[i][0] = 1.0

    x_train, x_test, y_train, y_test = train_test_split(regressor, regressand, test_size = 0.2)

    init_theta = np.array([np.ones(9)])
    theta = gradientDescentSingle(x_train, y_train, init_theta, 1.0e-3, 1.0e-12)
    pred_train = np.matmul(x_train, theta.T)
    pred_test = np.matmul(x_test, theta.T)

    R2 = [r2_score(y_train, pred_train), r2_score(y_test, pred_test)]
    MSE = [mean_squared_error(y_train, pred_train), mean_squared_error(y_test, pred_test)]
    print("R2 (single): ", R2)
    print("MSE (single): ", MSE, "\n")

    init_theta = np.array([np.ones(9)])
    theta = gradientDescent(x_train, y_train, init_theta, 1.0e-3, 1.0e-12)
    pred_train = np.matmul(x_train, theta.T)
    pred_test = np.matmul(x_test, theta.T)

    R2 = [r2_score(y_train, pred_train), r2_score(y_test, pred_test)]
    MSE = [mean_squared_error(y_train, pred_train), mean_squared_error(y_test, pred_test)]
    print("R2 (all): ", R2)
    print("MSE (all): ", MSE)

    print(" *** End of PTHREE\n\n")


def gradientDescentSingle(x, y, theta, alpha, precision):
    startTime = time.localtime(time.time())
    currentTime = time.localtime(time.time())
    prev_size = 1
    while(not(currentTime.tm_min >= startTime.tm_min+2 and currentTime.tm_sec >= startTime.tm_sec) and prev_size > precision):
        prev_theta = theta
        for j in range(len(theta[0])):
            secondTheta = 0
            for i in range(len(x[:, 0])):
                secondTheta = secondTheta + x[i][j]*(y[i][0] - np.matmul(x[i], theta[0].T))
            theta[0][j] = theta[0][j] + alpha*secondTheta

        prev_size = distance.euclidean(prev_theta[0], theta[0])
        currentTime = time.localtime(time.time())
    return theta
```

The code is basically the same as the last question except now we make a new gradient descent only update wj each iteration.That is ,instead of using matrix multiplication , we use for loop in every iteration in our previous gradient descent to compute cost gradient and update wj.

```
 *** Problem Three
R2 (single):  [0.3282035025118579, 0.32863496297319006]
MSE (single):  [0.6708441586588731, 0.6750115073996296]

R2 (all):  [0.6141302187199051, 0.6114939333771534]
MSE (all):  [0.38532277221242545, 0.3906162090691598]
 *** End of PTHREE
```

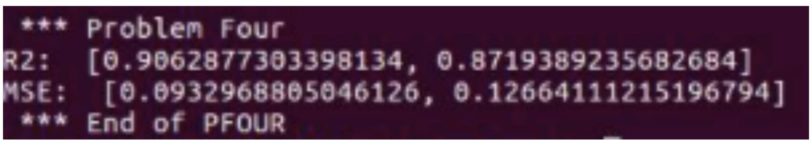
r2_score and mse for both training and testing datas.

(7)
Compare the performance between two different update method.
Ans:
It is obvious that updating single is less efficient than updating all since the r2_score is much lower and mse is much higher.It is because we set a time limit for the total iterations and updating single takes much longer time each iteration. So it is possible that updating single reaches time out and never get to the global minimum we want thus the result is not good as the original one.

(8)
The code, MSE, and the r2_score for problem 4
Ans:

```
def problemFour(concrete, regressand):
        print(" *** Problem Four")

        regressor = concrete.iloc[:, [0, 1, 2, 3, 4, 5, 6, 7]]
        regressor = regressor.astype(float)
        poly = PolynomialFeatures(3)
        regressor = poly.fit_transform(regressor)
        scaler = StandardScaler()
        regressor = scaler.fit_transform(regressor)
        for i in range(len(regressor)):
                regressor[i][0] = 1.0
        regressand = scaler.fit_transform(regressand)

        x_train, x_test, y_train, y_test = train_test_split(regressor, regressand, test_size = 0.2)

        init_theta = np.array([np.ones(165)])
        theta = gradientDescent(x_train, y_train, init_theta, 5.0e-5, 1.0e-6)
        pred_train = np.matmul(x_train, theta.T)
        pred_test = np.matmul(x_test, theta.T)

        R2 = [r2_score(y_train, pred_train), r2_score(y_test, pred_test)]
        MSE = [mean_squared_error(y_train, pred_train), mean_squared_error(y_test, pred_test)]
        print("R2: ", R2)
        print("MSE: ", MSE)

        print(" *** End of PFOUR")
```

We use all the features and fit with degree of 3 to do the polynomial regression.



```
 *** Problem Four
R2:  [0.9062877303398134, 0.8719389235682684]
MSE:  [0.0932968805046126, 0.1266411121519679]
 *** End of PFOUR
```

r2_score and mse for both training and testing datas.

(9)Answer the question

What is overfitting?
If training datas and iterations are not enough, or there are too many selecting features , the model will try to fit the training data and ignoring other possible data ,this may lead to high training accuracy but low testing accuracy, called overfitting.

Stochastic gradient descent is also a kind of gradient descent, what is the benefit of using SGD?
The benefit of using SGD is that it is faster because it pick one random data to compute gradient and update theta instead of pick all of the datas.Overall it is going at the direction of global minimum as the original GD version but it may stuck in local minimum if there is noise.

Why the different initial value to GD model may cause different result?
If the original value is not appropriate(may be far from global minimum), it may stuck in local minimum and never gets to the global minimum.

What is the bad learning rate? What problem will happen if we use it?
Bad learning rate is basically learning rate that is too high or too low.Like alpha in the gradient descent, if it is too small, it takes more iterations and more time to get global minimum ,thus less efficient, if it is too big, theta may swing left and right of global minimum and can't get to that value.

After finishing this homework, what have you learned, what problems you encountered, and how the problems were solved?
We have learned how to use basic linear and polynomial regression model and design an efficient gradient descent computing function.The biggest problem is the bonus one, first we only use 2 features and degree of 2, but we cannot get r2_score>0.87, so we put all the features in and raise the degree to 3 and we get what we want.

(10)
Bonus
Ans:

```
def problemFour(concrete, regressand):
	print(" *** Problem Four")

	regressor = concrete.iloc[:, [0, 1, 2, 3, 4, 5, 6, 7]]
	regressor = regressor.astype(float)
	poly = PolynomialFeatures(3)
	regressor = poly.fit_transform(regressor)
	scaler = StandardScaler()
	regressor = scaler.fit_transform(regressor)
	for i in range(len(regressor)):
		regressor[i][0] = 1.0
	regressand = scaler.fit_transform(regressand)

	x_train, x_test, y_train, y_test = train_test_split(regressor, regressand, test_size = 0.2)

	init_theta = np.array([np.ones(165)])
	theta = gradientDescent(x_train, y_train, init_theta, 5.0e-5, 1.0e-6)
	pred_train = np.matmul(x_train, theta.T)
	pred_test = np.matmul(x_test, theta.T)
```

```
R2 = [r2_score(y_train, pred_train), r2_score(y_test, pred_test)]
MSE = [mean_squared_error(y_train, pred_train), mean_squared_error(y_test, pred_test)]
print("R2: ", R2)
print("MSE: ", MSE)

print(" *** End of PFOUR”)
```

Since we did not keep the original version of p4 this is the updated version for bonus question ,the same as code in question8.We use all the features and fit with degree of 3 to do the polynomial regression.And we get r2_score>0.87.

```
 *** Problem Four
R2:  [0.9062877303398134, 0.8719389235682684]
MSE:  [0.0932968805046126, 0.12664111215196794]
 *** End of PFOUR
```

r2_score and mse for both training and testing datas.