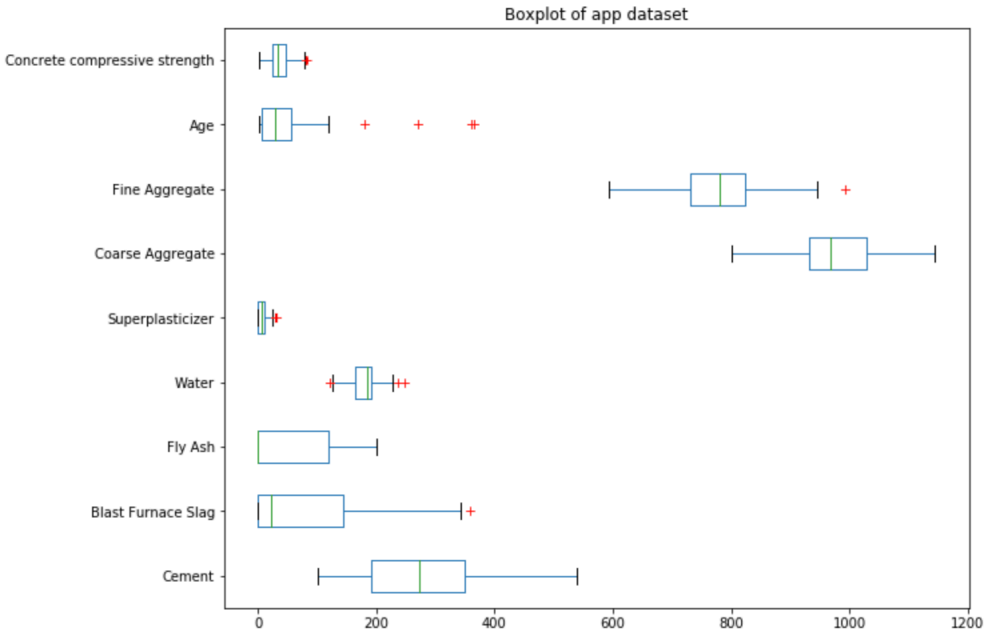
Introduction to Machine Learning Assignment #3

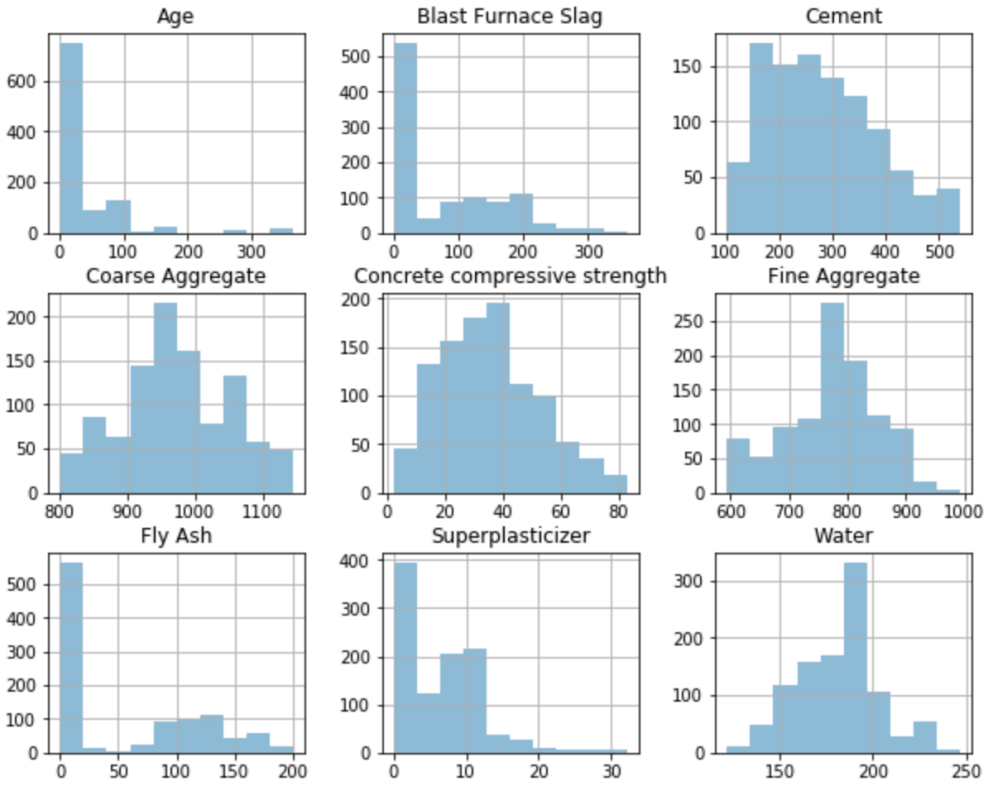
Team ID: 15
Team member: 0416004 郭羽喬 0416208 黃士軒 0416025 呂翊愷 0416022 楊旻學

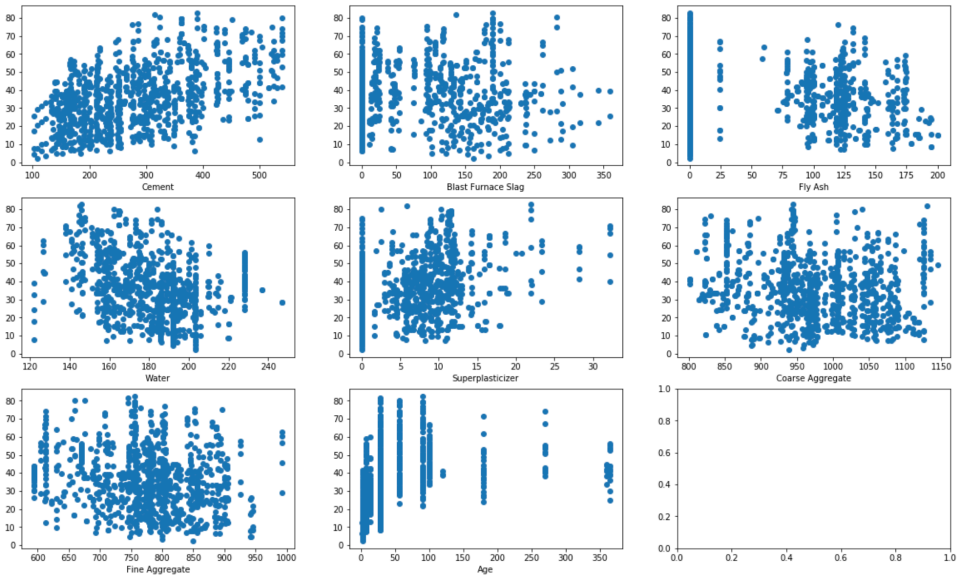
1. What environments the members are using

   NCHC-TWGC Machine Learning Cloud service with V100 GPU

2. Visualization of all the features with the target







3. The code, graph, r2_score, weight and bias for problem 1

```
numInstances = len(concreteDataset)
X = concreteDataset['Cement'].values.reshape((numInstances, 1))
Y = concreteDataset['Concrete compressive strength'].values
```

By visualizing all the features of the target, we discover that cement is the most relative feature, so we use cement as data instance and concrete compressive strength as target.

```
xTrain, xTest, yTrain, yTest = train_test_split(X, Y, test_size=0.2, random_state=0)
print('#Training data points: %d' % xTrain.shape[0])
print('#Testing data points: %d' % xTest.shape[0])

# Training
reg = LinearRegression()
reg.fit(xTrain, yTrain)

# Testing
yTrainPredict = reg.predict(xTrain)
yTestPredict = reg.predict(xTest)
```

We split 8/10 dataset to be the training set and the other 2/10 dataset to be the testing set by using `sklearn.model_selection.train_test_split`. Training set is used to build the training model, and testing set is used to evaluate the performance of the prediction model in final. Then we use `sklearn.linear_model.LinearRegression` and `fit` to build and train the linear regression model. With the linear regression model, we can use the model to get the prediction of testing dataset by `predict`.

```
print('Slope (w_1): %.2f' % reg.coef_[0])
print('Intercept/bias (w_0): %.2f' % reg.intercept_)
print('MSE train: %.2f, test: %.2f' %
      (mean_squared_error(yTrain, yTrainPredict),
       mean_squared_error(yTest, yTestPredict)))
print('R^2 train: %.2f, test: %.2f' % (r2_score(yTrain, yTrainPredict),
                                       r2_score(yTest, yTestPredict)))
```

Finally, we get the linear equation of the concrete dataset. By using `sklearn.metrics.mean_squared_error` function, we can calculate mean squared error of real target and predicted target. By using `sklearn.metrics.r2_score` function, we can calculate r2_score of real target and predicted target.

```
#Training data points: 824
#Testing data points: 206
Slope (w_1): 0.08
Intercept/bias (w_0): 14.66
MSE train: 217.84, test: 178.51
R^2 train: 0.23, test: 0.32
```

The above image is our result of problem 1. It can be seen that our r2_score of training data is 0.23, and r2_score of testing data is 0.32. The weight of our linear regression model is 0.08, and the bias of our linear regression model is 14.66.

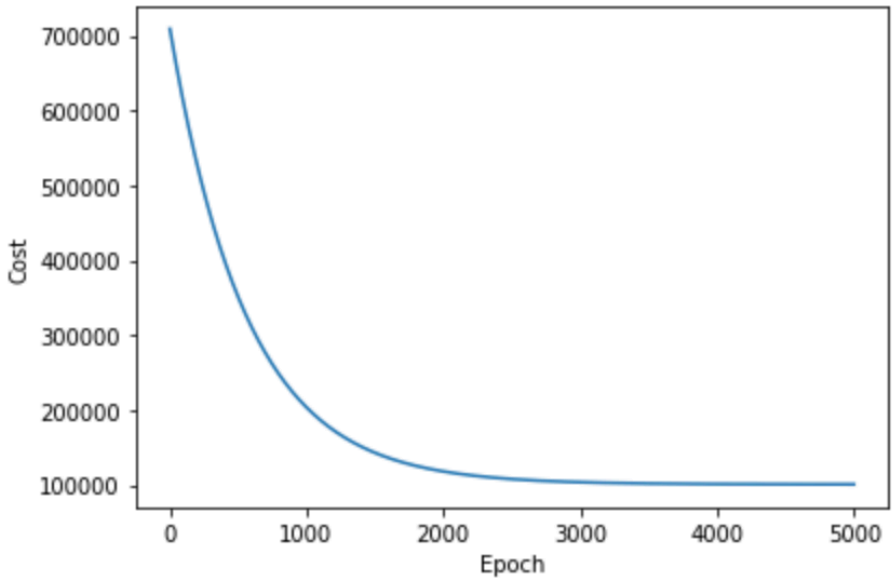
4. The code, graph, r2_score, weight and bias for problem 2

```
class LinearRegressionGradientDescent(object):
    def __init__(self, learningRate=0.00000001, numInterations=5000, randomState=1, updateOption=0): #If
        self.learningRate = learningRate
        self.numInterations = numInterations
        self.randomState = randomState
        self.updateOption = updateOption

    def fit(self, x, y):
        rgen = np.random.RandomState(self.randomState)
        self.theta_ = rgen.normal(loc=0.0, scale=0.01, size=1 + x.shape[1])
        self.numInstances_ = x.shape[0]
        self.cost_ = []
        if self.updateOption == 0:
            for i in range(self.numInterations):
                prediction = self.network(x)
                errors = prediction - y
                # Cost function
                cost = (errors**2).sum() / 2
                self.cost_.append(cost)
                # Update the theta
                self.theta_[1:] -= self.learningRate * (x.T.dot(errors) / self.numInstances_)
                self.theta_[0] -= self.learningRate * (errors.sum() / self.numInstances_)
                #print(self.theta_)
            return self
        else:
            for i in range(self.numInterations):
                prediction = self.network(x)
                errors = prediction - y
                # Cost function
                cost = (errors**2).sum() / 2
                self.cost_.append(cost)
                # Update the theta
                idx = i % (len(self.theta_) - 1)
                if idx == 0: self.theta_[0] -= self.learningRate * (errors.sum() / self.numInstances_)
                else: self.theta_[idx] -= self.learningRate * (x.T.dot(errors)[idx] / self.numInstances_)
                #print(self.theta_)
            return self
```

```
reg = LinearRegressionGradientDescent()
reg.fit(xTrain, yTrain)

# Testing
yTrainPredict = reg.predict(xTrain)
yTestPredict = reg.predict(xTest)
```

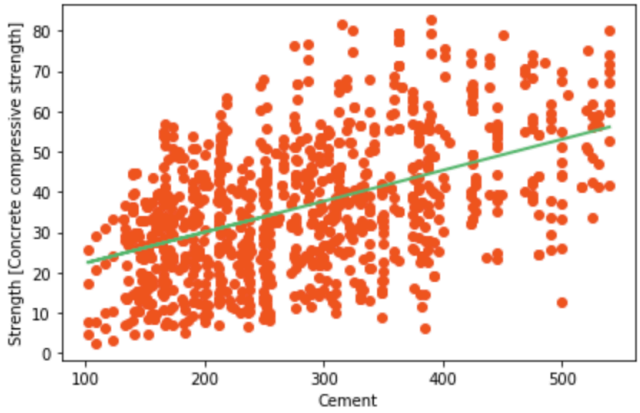
To implement gradient descent, we randomly choose initial weight and bias to calculate loss function every iteration. We calculate the dot of data and weight, and the MSE of prediction and ground truth. Afterward, we update the weight and iterate the above steps 5000 times, and finally we can find the best weight of each instance, which can help us to get linear regression.

```
#Training data points: 824
#Testing data points: 206
Slope (w_1): 0.12
Intercept/bias (w_0): 0.02
MSE train: 244.43, test: 180.20
R^2 train: 0.13, test: 0.32
[100703.29458585984]
```

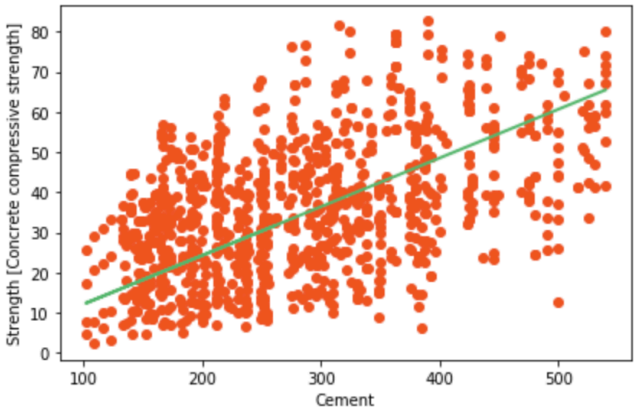


The above image is our result of problem 2. It can be seen that our r2_score of training data is 0.13, and r2_score of testing data is 0.32. The weight of our linear regression model is 0.12, and the bias of our linear regression model is 0.02. Besides, by the curve diagram, we discover that the cost will gradually decrease by more iterations, so it seems that our gradient descent is indeed useful.

5. Compare Problem1 and Problem2, show what you got



(a) linear regression result of problem1



(b) linear regression result of problem2

The weight and the bias we get in problem1 and problem2 is approximately the same. However, when we implement problem2, we discover that learning rate is highly relevant to gradient. If learning rate is too small, convergence will be slow. If learning rate is too large, theta may not decrease on every iteration and also may not converge. As a result, we choose learning rate many times to find the better one, and finally we get the best result of our own gradient, which is also the closest result compared to the linear regression of using built-in function.

6. The code, MSE, and the r2_score for problem 3

```
# Standardization
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)

y_train_std = preprocessing.scale(y_train)
y_test_std = preprocessing.scale(y_test)
```

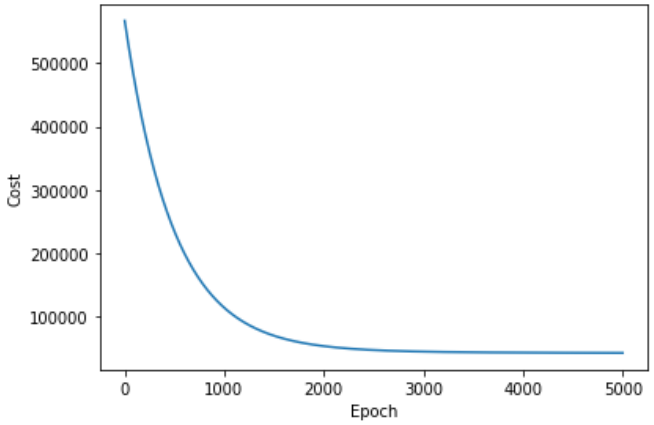
Due to the scale of each feature in original dataset is different, we need to preprocess the data by standardization, or different feature will cause different influence of prediction. Therefore, we use `sklearn.preprocessing.transform` to standardize both training data and test data.

```
# Training
reg = LinearRegressionGradientDescent(0.001, 5000, 1, 0)
reg.fit(X_train_std, y_train)
print(reg.theta_)

# Testing
y_train_pred = reg.predict(X_train_std)
y_test_pred = reg.predict(X_test_std)
```

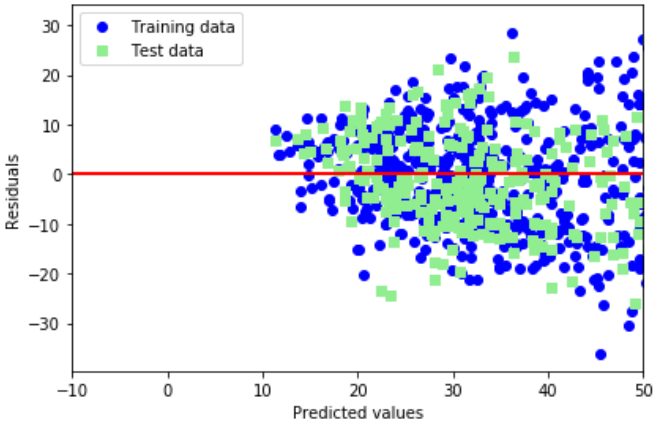
Different from problem2, we choose the higher learning rate, since we find that the learning rate in problem2 is not enough to converge the data. In every iteration, we update all the weight. Then we use `fit to` train the standardized data and `predict` to get the prediction of the training model.

```
#Training data points: 721
#Testing data points: 309
[ 35.61282183   7.26046311   4.00722921   0.71873043  -5.43547091
   3.356733    -1.35571571  -2.64295351   6.75328404]
MSE train: 118.70, test: 97.54
R^2 train: 0.59, test: 0.62
```

The above image is our result of problem 3. It can be seen that our r2_score of training data is 0.59, and r2_score of testing data is 0.62, MSE of training data is 118.70, and MSE of testing data is 97.54. R2_score is higher than problem1 and problem2, and MSE is lower than problem1 and problem2, so we think that multi-variable is indeed helpful of higher accuracy and lower error.



(c) cost in different numbers of iteration



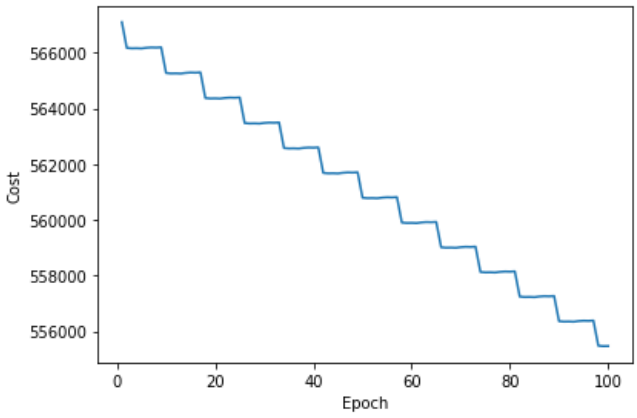
(d) residuals of predicted value and original value of each predicted values

By the curve diagram (c), we discover that cost will gradually decrease by more iterations, so it seems that our gradient descent is indeed useful. Besides, the diagram (d) shows that the difference between our predicted value and original value in every predicted values is small, which means that the loss of our prediction is low.

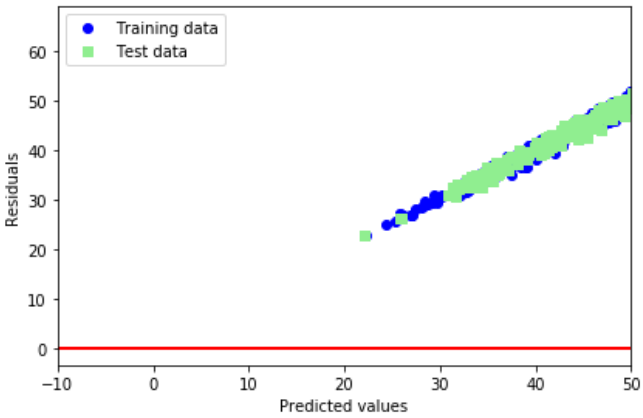
7. Compare the performance between two different update method

In problem 3, we also implement the version of updating only one weight each iteration.

```
[ 0.47934515  0.0272802  -0.03045922 -0.07672637  0.08531261 -0.05541449
 -0.01583382  0.05757416  0.00319039]
MSE train: 449.16, test: 416.36
R^2 train: -0.56, test: -0.62
```



(e) cost in different numbers of iteration



(f) residuals of each predicted values

The performance of updating only one weight each iteration is much worse than update all the weight each iteration. Not only MSE is much higher but also R2 score is even lower to negative number. If R2 score is negative number, it means that the prediction is not accurate at all. Besides, by figure (f), the cost is also much higher. As a result, we think that updating all the weight each iteration has better performance.

8. The code, MSE, and the r2_score for problem 4

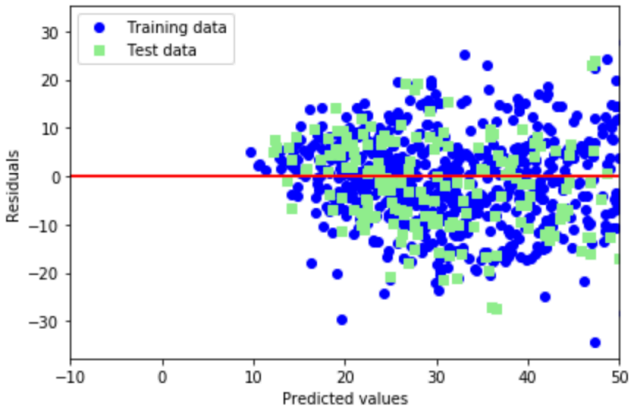
```
# Standardization
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)

quadratic = PolynomialFeatures(2)
X_train_quad = quadratic.fit_transform(X_train_std)
X_test_quad = quadratic.fit_transform(X_test_std)

cubic = PolynomialFeatures(3)
X_train_cubic = cubic.fit_transform(X_train_std)
X_test_cubic = cubic.fit_transform(X_test_std)

quartic = PolynomialFeatures(4)
X_train_quar = quartic.fit_transform(X_train_std)
X_test_quar = quartic.fit_transform(X_test_std)
```

```
# Training
reg_quad = LinearRegressionGradientDescent(learningRate=0.001, numInterations=5000)
reg_quad.fit(X_train_quad, y_train)

reg_cubic = LinearRegressionGradientDescent(learningRate=0.001, numInterations=5000)
reg_cubic.fit(X_train_cubic, y_train)

reg_quar = LinearRegressionGradientDescent(learningRate=0.001, numInterations=5000)
reg_quar.fit(X_train_quar, y_train)

# Testing
y_train_pred_quad = reg_quad.predict(X_train_quad)
y_test_pred_quad = reg_quad.predict(X_test_quad)

y_train_pred_cubic = reg_cubic.predict(X_train_cubic)
y_test_pred_cubic = reg_cubic.predict(X_test_cubic)

y_train_pred_cubic = reg_cubic.predict(X_train_cubic)
y_test_pred_cubic = reg_cubic.predict(X_test_cubic)
```

Due to multiple variables, we standardize training data and test data first. We use `sklearn.preprocessing.PolynomialFeatures` to generate a feature matrix consisting of all polynomial combinations of the features, and use `fit_transform` to fit transformer to data and transform it. By doing so, we build the polynomial model with degree 2, 3, 4 respectively. Then we train and test the polynomial model by our own gradient descent with learning rate 0.001 and 5000 iterations.

```
quadratic:
theta:
[  1.41247149e+01   1.41023539e+01   6.57120547e+00   2.63507312e+00
  -1.19317763e+00  -4.37904181e+00   3.77783783e+00  -1.20788388e+00
  -1.23912221e+00   1.01551733e+01   1.63787462e+00  -2.09320947e-01
  -1.39407497e-03  -2.94942093e+00   5.98536715e-01  -7.09459532e-01
  -6.60834043e-01  -3.95391097e-01   1.69350499e+00  -7.60963928e-01
  -1.86102045e+00   2.00044883e+00  -1.36148353e+00   3.46339672e-02
   5.38558394e-01   2.20266069e+00  -5.24277833e-01   9.78140665e-01
   5.87439968e-01   2.00186325e+00  -2.01613210e-02   1.22703643e+00
  -1.99199807e+00   1.52733664e-01  -3.17923310e-01  -6.96960655e-01
  -2.75579313e+00   1.05885186e+00  -4.32621848e-01   3.31814783e-01
   2.11061975e+00   1.86947085e+00   1.20007716e+00  -2.13430147e-01
   4.99391522e-01  -1.26461247e+00]
MSE train: 91.41, test: 99.47
R^2 train: 0.68, test: 0.62
```

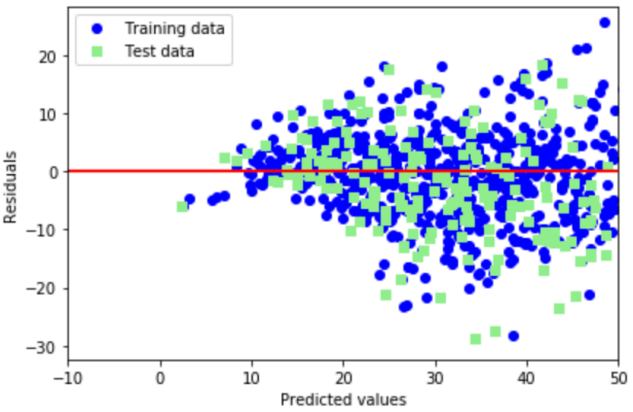
(g) theta & MSE & R2 of quadratic



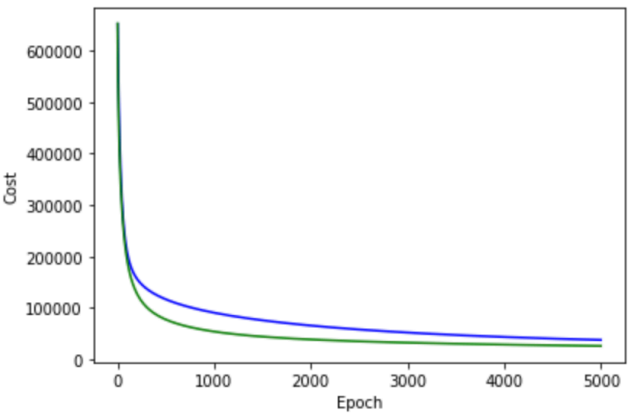
(h) residuals of each predicted values of quadratic

```
cubic:
theta:
[ 11.05562233  11.03326132   1.1302518    1.598842     0.80177966
  -1.75629127   1.96401605  -0.53148286  -1.09424339   3.6205804
   2.24263099  -0.80721408  -1.83982741  -1.0570726    0.66686722
  -0.89295635   0.43147338  -0.28076848   4.11903679  -2.12360812
  -1.12186588   0.28869859  -1.24483704  -0.45936132   0.724029
   5.97667806   0.09133261   0.39931259   0.37441675   0.29985787
  -0.26896017   2.22230587  -1.93829432  -0.54844036  -0.34898099
  -0.8281704    0.37453613   0.56156775   0.29511332  -0.79696059
   3.21702002   0.90354897   0.66226349  -0.10877587   1.69306696
  -1.83029601   0.45722611   1.53436876  -0.55557658  -0.7115301
  -0.18134286   0.23638411   0.35614023   1.25476837   2.52018435
  -2.17476841   0.41350158  -0.28764432  -1.44320508  -0.29644397
   1.03377746   2.33805934   0.21770604   1.01084212  -0.85455201
  -0.68873886  -0.0875608    1.08622786  -0.07632674  -0.69614675
   0.46626658  -0.119671     0.35865471   0.14123526  -0.4348133
  -0.75637355   0.06653309   0.31294045  -1.01085532   0.43353688
   0.30513409   0.02189144  -0.84516982  -0.38689919  -0.93077203
   0.25381095   0.30174097  -0.73669192   2.25464027   1.36695112
  -0.40329305   0.12678192   1.29017383  -0.10600038  -2.49122283
   0.52457974  -0.0736125   -0.14187964   0.46192755  -1.25269635
   0.0313823    0.01577004  -0.11523633   0.40837502   1.64103128
   0.22755039  -0.46391432  -0.54587911  -0.31298556  -0.26715574
  -1.14699971  -0.88941281   0.13471639   0.66077513  -0.02603178
   3.31466606  -0.17230409  -0.78203155   1.32804225  -0.43934187
  -0.3634075   -0.12329096  -0.50122342  -0.46004062   1.81009545
   0.29101094  -0.59659965  -0.30166523   1.36559003  -0.04579695
   1.43320794   0.70029241   1.26866995   0.16089422   0.20537436
   0.88002954   0.46986927   0.15664008  -0.43818027  -0.94372952
  -0.98711179  -0.57738299   0.78485206  -0.11856159   0.42582777
  -0.18246803  -1.09239002  -0.99908076  -0.04675516   0.63549655
   0.23775734   1.42663776   0.23353509  -0.3362697   -0.39756689
   0.91801437  -0.24458936  -0.47620327   1.52936524  -0.08856214
  -0.11787871  -0.48643812   0.27077785  -0.49333902  -0.29037368
   0.41927262]
MSE train: 62.91, test: 82.83
R^2 train: 0.78, test: 0.69
```

(i) theta & MSE & R2 of cubic



(j) residuals of each predicted values of cubic



(j) residuals of each predicted values of cubic

The performance of polynomial regression is much better than linear regression. Not only MSE is much lower but also R2 score is much higher. By figure (f), the cost is dramatically decreased, so it illustrates that polynomial regression can enable the model to

converge faster. Hence, it seems that polynomial regression may fit the data more than linear regression.

9. Answer the question
   A. What is overfitting?
      Overfitting means that using too many parameters or using too complex function when training a model. It leads to fit the data (including noise) too closely, and further fail to predict test data reliably.

   B. Stochastic gradient descent is also a kind of gradient descent, what is the benefit of using SGD?
      Since SGD randomly choose an instance to update the weight of model, it may enable to optimize the weight from the current local minimum to another better local minimum faster, and finally get global minimum.

   C. Why the different initial value to GD model may cause different result?
      GD model can only find local minimum, not certainly global minimum. Different initial value may cause GD model to find different minimum, i.e. different result.

   D. What is the bad learning rate? What problem will happen if we use it?
      Bad learning rate means that using too high or low learning rate. If learning rate is too high, the loss function may not be able to converge. If learning rate is too low, the loss function may converge too slowly.

   E. After finishing this homework, what have you learned, what problems you encountered, and how the problems were solved?
      We learned how to use `sklearn` to implement standardization, linear regression, MSE, R2. Furthermore, we implement our own gradient descent for linear regression and polynomial regression. Since there always some gaps between theorem and code implementation, we have a heave discuss on how to make the code run and build efficiently, and after refactoring the code again and again, we successfully make the code run and be as general as possible.

10. Bonus

```
class LinearRegressionGradientDescentWithLogCosh(object):
    def __init__(self, learningRate=0.00000001, numInterations=5000, randomState=15):
        self.learningRate = learningRate
        self.numInterations = numInterations
        self.randomState = randomState

    def fit(self, x, y):
        rgen = np.random.RandomState(self.randomState)
        self.theta_ = rgen.normal(loc=0.0, scale=0.01, size=1 + x.shape[1])
        self.numInstances_ = x.shape[0]
        self.cost_ = []
        for i in range(self.numInterations):
            prediction = self.network(x)
            errors = prediction - y
            # Cost function
            cost = np.log(np.cosh(errors)).sum()
            self.cost_.append(cost)
            # Update the theta
            errors_sinh = np.sinh(errors)
            errors_cosh = np.cosh(errors)
            errors_tmp = np.true_divide(errors_sinh, errors_cosh)
            self.theta_[1:] -= self.learningRate * x.T.dot(errors_tmp)
            self.theta_[0] -= self.learningRate * errors_tmp.sum()

        return self

    def network(self, x):
        return np.dot(x, self.theta_[1:]) + self.theta_[0]

    def predict(self, x):
        return self.network(x)
```

```
cubic = PolynomialFeatures(3)
X_train_cubic = cubic.fit_transform(X_train_std)
X_test_cubic = cubic.fit_transform(X_test_std)

# Training
reg_cubic = LinearRegressionGradientDescentWithLogCosh(learningRate=0.00003, numInterations=15000)
reg_cubic.fit(X_train_cubic, y_train)

# Testing
y_train_pred_cubic = reg_cubic.predict(X_train_cubic)
y_test_pred_cubic = reg_cubic.predict(X_test_cubic)
```

We change our original loss function to Log-cosh loss function. The advantage of log-cosh loss is that it is less likely affected by noise. Besides, we use cubic polynomial, which can make cost function lower.

```
MSE train: 29.13, test: 47.18
R^2 train: 0.90, test: 0.82
```