Problem 1:

(a) False. Ridge Regression has a closed form solution:

$$\begin{aligned}
J(\theta) &= \frac{1}{2n}\sum_{i=1}^{n}(h_\theta(x^{(i)})-y^{(i)})^2+\frac{\lambda}{2}\sum_{k=1}^{d}\theta_k^2\\
&= \frac{1}{2n}\sum_{i=1}^{n}(\theta^T x^{(i)}-y^{(i)})^2+\frac{\lambda}{2}\theta\theta^T\\
&= \frac{1}{2n}(X\theta-y)(X\theta-y)^T+\frac{\lambda}{2}\theta\theta^T\\
&= \frac{1}{2n}\cdot(X\theta\theta^TX^T-y\theta^TX^T-X\theta y^T+yy^T)+\frac{\lambda}{2}\theta\theta^T\\
&= \frac{1}{2n}\cdot(X\theta\theta^TX^T-2y\theta^TX^T+yy^T)+\frac{\lambda}{2}\theta\theta^T
\end{aligned}$$

Then, $\frac{\partial}{\partial\theta}J(\theta)=\frac{1}{2n}\frac{\partial}{\partial\theta}(X\theta\theta^TX^T-2y\theta^TX^T+yy^T)+\frac{\lambda}{2}\frac{\partial}{\partial\theta}\theta\theta^T=0$

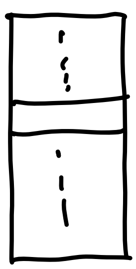
$$\Rightarrow \frac{1}{2n}(2X^TX\theta-2X^Ty)+\lambda\theta=0$$

$$\left(\frac{1}{n}X^TX+\lambda I_{d+1}\right)\theta-\frac{1}{n}X^Ty=0$$

$$\theta=(X^TX+n\lambda\cdot I_{d+1})^{-1}X^Ty$$

I_{d+1} is $(d+1)$ by $(d+1)$ identity matrix.

(b) True.

k-folds  with m values of a hyperparameter.

Then we will test k times for each value of the hyperparameter. Hence, In total, we need mk times.

(c). True

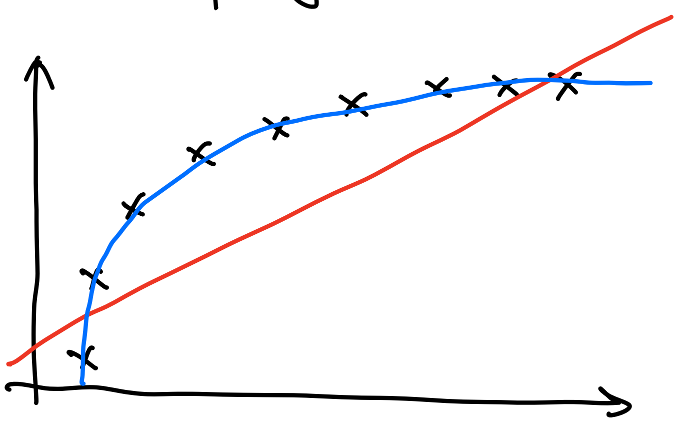
$$
\begin{aligned}
MSE &= \frac{1}{n}\sum_{i=1}^{n}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(\theta^T \phi^{(i)}(x) - y^{(i)}\right)^2 \\
&= \frac{1}{n}\left(\phi(x)\theta^T - y\right)\left(\phi(x)\theta^T - y\right)^T \\
&= \frac{1}{n}\left(\phi(x)\theta^T\theta\,\phi(x)^T - y\,\theta\,\phi(x)^T - \phi(x)\theta^T y^T - y y^T\right) \\
&= \frac{1}{n}\left(\phi(x)\theta^T\theta\,\phi(x)^T - 2y\,\theta\,\phi(x)^T - y y^T\right)
\end{aligned}
$$

Then, want $\frac{\partial}{\partial\theta}(MSE) = 0$; namely we want

$$
\begin{aligned}
\frac{\partial}{\partial\theta}\left(\phi(x)\theta^T\theta\,\phi(x)^T - 2y\,\theta\,\phi(x)^T - y y^T\right) &= 0 \\
\Rightarrow \phi(x)^T\phi(x)\theta - \phi(x)^T y &= 0 \\
\Rightarrow \theta &= \left(\phi(x)^T\phi(x)\right)^{-1}\phi(x)^T y
\end{aligned}
$$

(d). A.

The relationship may be non-linear. For example:



we see that the red line is a linear relation ship, but the true relationship is non-linear (blue curve)

(e). C.

Loss Regularization is a method for preventing over-fitting by automatically controlling the complexity of the learned hypothesis. It penalize large values of θ_j during optimization.

Problem 2:

(a)

$$J_{MAE}(\theta) = \frac{1}{n}\sum_{i=1}^{n} |\theta^T x^{(i)} - y^{(i)}|$$

$$= \frac{1}{n}\sum_{i=1}^{n} \text{sign}(\theta^T x^{(i)} - y^{(i)}) \cdot (\theta^T x^{(i)} - y^{(i)})$$

so $\frac{\partial}{\partial\theta_j} J_{MAE}(\theta) = \frac{\partial}{\partial\theta_j}\left(\frac{1}{n}\sum_{i=1}^{n} \text{sign}(\theta^T x^{(i)} - y^{(i)}) \cdot (\theta^T x^{(i)} - y^{(i)})\right)$

$$= \frac{1}{n}\sum_{i=1}^{n} \text{sign}(\theta^T x^{(i)} - y^{(i)}) \frac{\partial}{\partial\theta_j}(\theta^T x^{(i)} - y^{(i)})$$

$$= \frac{1}{n}\sum_{i=1}^{n} \text{sign}(\theta^T x^{(i)} - y^{(i)}) \cdot \frac{\partial}{\partial\theta_j}\left(\sum_{k=1}^{d} \theta_k x_k^{(i)} - y_k^{(i)}\right)$$

$$= \frac{1}{n}\sum_{i=1}^{n} \text{sign}(\theta^T x^{(i)} - y^{(i)}) \cdot x_j^{(i)}$$

(b).

$$\nabla_\theta J_{MAE}(\theta) = \begin{pmatrix} \frac{\partial J_{MAE}(\theta)}{\partial\theta_1} \\ \frac{\partial J_{MAE}(\theta)}{\partial\theta_2} \\ \vdots \\ \frac{\partial J_{MAE}(\theta)}{\partial\theta_d} \end{pmatrix} = \begin{pmatrix} \frac{1}{n}\sum_{i=1}^{n} \text{sign}(\theta^T x^{(i)} - y^{(i)}) \cdot x_1^{(i)} \\ \vdots \\ \frac{1}{n}\sum_{i=1}^{n} \text{sign}(\theta^T x^{(i)} - y^{(i)} \cdot x_d^{(i)} \end{pmatrix}$$

$$= \frac{1}{n} \underset{n\times(d+1)}{X^T} \cdot \text{sign}(X\underset{(d+1)\cdot 1}{\theta} - y)$$

(c).

$$J_{MAE}(\theta) = \frac{1}{n}\sum_{i=1}^{n} |\theta^T x^{(i)} - y^{(i)}|$$

$$= \frac{1}{n} \cdot \sum_{i=1}^{n} \left| \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} x^{(i)} - y^{(i)} \right|$$

$$= \frac{1}{n} \cdot \sum_{i=1}^{n} \left| x_0^{(i)} + x_1^{(i)} + x_2^{(i)} - y^{(i)} \right|$$

$$= \frac{1}{8} \cdot \big((12-1-4) + (3-1-1-1) + (1+1-1) + (6-1+2+2) + (3-1+2-1) + (6-1-1) + (8-5-2-1) + (7-3-1)\big)$$

$$= \frac{1}{8} \cdot (7+1+9+3+4+3)$$

$$= \frac{27}{8}$$

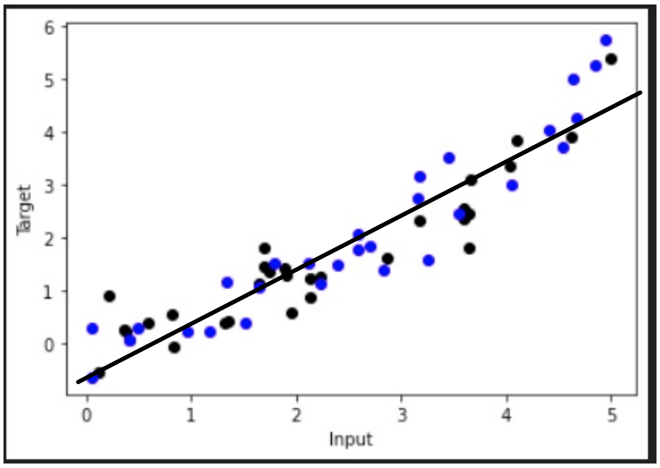
$$
\begin{aligned}
J_{MSE}(\theta) &= \frac{1}{n}\sum_{i=1}^{n}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} - y^{(i)}\right)^2 \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(x_0^{(i)} + x_1^{(i)} + x_2^{(i)} - y^{(i)}\right)^2 \\
&= \frac{1}{8}\left((12-1-4)^2 + (3-1-1-1)^2 + (1+1-1)^2 + (6-1+2+2)^2 + (3-1+2-1)^2 + (6-1-1)^2 + (8-5-2-1)^2 + (7-3-1)^2\right) \\
&= \frac{1}{8}\left(7^2 + 1^2 + 9^2 + 3^2 + 4^2 + 3\right) \\
&= \frac{1}{8}\left(49 + 1 + 81 + 9 + 16 + 9\right) \\
&= \frac{1}{8}\cdot 165 \\
&= \frac{165}{8}
\end{aligned}
$$

$$
\begin{aligned}
\nabla_\theta J_{MAE}(\theta) &= \frac{1}{n} X^T \cdot \mathrm{sign}(X\theta - y) \\
&= \frac{1}{n}\begin{bmatrix} 1 & \cdots & 1 \\ x_1^{(1)} & \cdots & x_1^{(n)} \\ x_2^{(1)} & \cdots & x_2^{(n)} \end{bmatrix} \cdot \mathrm{sign}\left(\begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} \\ \vdots & \vdots & \vdots \\ 1 & x_1^{(n)} & x_2^{(n)} \end{bmatrix}\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{bmatrix}\right) \\
&= \frac{1}{8}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 4 & 1 & 0 & -2 & -2 & 1 & 5 & 3 \\ 0 & 1 & 1 & -2 & 1 & 0 & 2 & 0 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \\ -1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \\
&= \frac{1}{8}\begin{bmatrix} -2 \\ 2 \\ 5 \end{bmatrix} \\
&= \begin{bmatrix} -0.25 \\ 0.25 \\ 0.625 \end{bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
\nabla_\theta J_{MSE}(\theta) &= \nabla_\theta \frac{1}{n}(X\theta - y)^T(X\theta - y) \\
&= \frac{1}{n}\nabla_\theta\left(\theta^T X^T X\theta - 2\theta^T X^T y + y^T y\right) \\
&= \frac{1}{n}\nabla_\theta\left(\mathrm{tr}\,\theta^T X^T X\theta - 2\,\mathrm{tr}\,\theta^T X^T y\right) \\
&= \frac{2}{n}\left(X^T X\theta - X^T y\right) \\
&= \frac{2}{8}\left(\begin{bmatrix} 21 \\ 83 \\ 27 \end{bmatrix} - \begin{bmatrix} 46 \\ 58 \\ 11 \end{bmatrix}\right) \\
&= \frac{2}{8}\begin{bmatrix} -25 \\ 25 \\ 16 \end{bmatrix} = \begin{bmatrix} -6.25 \\ 6.25 \\ 4 \end{bmatrix}
\end{aligned}
$$

Problem 3:

(a)



From the plot, we see that it's mostly lineary, but with the right end slightly tailed upwards. So there might be a slight skewness in the dataset.

(b). get_poly_features()

```
def get_poly_features(self, X):
    """
    Inputs:
     - X: A numpy array of shape (n,1) containing the data.
    Returns:
     - X_out: an augmented training data as an mth degree feature vector
     e.g. [1, x, x^2, ..., x^m], x \in X.
    """
    n,d = X.shape
    m = self.m
    X_out= np.zeros((n,m+1))
    if m==1:
        # ================================================================ #
        # YOUR CODE HERE:
        # IMPLEMENT THE MATRIX X_out with each entry = [1, x]
        # ================================================================ #
        for i in range(0, n):
            X_out[i, :] = np.array([1, X[i, 0]])
        pass
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
    else:
        # ================================================================ #
        # YOUR CODE HERE:
        # IMPLEMENT THE MATRIX X_out with each entry = [1, x, x^2,....,x^m]
        # ================================================================ #
        for i in range(0, n):
            for j in range(0, m+1):
                X_out[i, j] = pow(X[i, 0], j)
        pass
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
        pass
    return X_out
```

(c) predict()

```python
def predict(self, X):
    """
    Inputs:
    - X: n x 1 array of training data.
    Returns:
    - y_pred: Predicted targets for the data in X. y_pred is a 1-dimensional
      array of length n.
    """
    y_pred = np.zeros(X.shape[0])
    m = self.m
    theta = self.theta
    if m==1:
        # ================================================================ #
        # YOUR CODE HERE:
        # PREDICT THE TARGETS OF X
        # ================================================================ #
        for i in range(0, X.shape[0]):
            y_pred[i] = theta[0] + theta[1] * X[i]
        pass
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
    else:
        # ================================================================ #
        # YOUR CODE HERE:
        # Extend X with get_poly_features().
        # Predict the target of X.
        # ================================================================ #
        polyX = self.get_poly_features(X)
        for i in range(0, polyX.shape[0]):
            for j in range(0, polyX.shape[1]):
                y_pred[i] += theta[j] * polyX[i, j]
        pass
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
    return y_pred
```

part (c) test:

```python
## PART (c):
## Complete loss_and_grad function in Regression.py file and test your results.
regression = Regression(m=1, reg_param=0)
loss, grad = regression.loss_and_grad(X_train,y_train)
print('Loss value',loss)
print('Gradient value',grad)

##
```

✓ 0.0s

```
Loss value 1.0455416122950605
Gradient value [1.33142275 2.65167278]
```

(d) train_LR().

```python
def train_LR(self, X, y, alpha=1e-2, B=30, num_iters=10000) :
    """
    Finds the coefficients of a {d-1}^th degree polynomial
    that fits the data using least squares mini-batch gradient descent.

    Inputs:
     - X          -- numpy array of shape (n,d), features
     - y          -- numpy array of shape (n,), targets
     - alpha      -- float, learning rate
     -B           -- integer, batch size
     - num_iters -- integer, maximum number of iterations

    Returns:
     - loss_history: vector containing the loss at each training iteration.
     - self.theta: optimal weights
    """
    ### These two lines set the random seeds... you can ignore. #####
    random.seed(10)
    np.random.seed(10)
    #################################################################
    self.theta = np.random.standard_normal(self.dim)
    loss_history = []
    n,d = X.shape
    for t in np.arange(num_iters):
        X_batch = None
        y_batch = None
        # ================================================================ #
        # YOUR CODE HERE:
        # Shuffle X, y along the batch axis with np.random.shuffle.
        # Get the first batch_size elements X_batch from X, y_batch from Y.
        # X_batch should have shape: (B,1), y_batch should have shape: (B,).
        # ================================================================ #
        indices = np.arange(n)
        np.random.shuffle(indices)
        X = X[indices]
        y = y[indices]
        X_batch = X[0:B]
        y_batch = y[0:B]
        pass
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
        loss = 0.0
        grad = np.zeros_like(self.theta)
        # ================================================================ #
        # YOUR CODE HERE:
        # evaluate loss and gradient for batch data
        # save loss as loss and gradient as grad
        # update the weights self.theta
        # ================================================================ #
        loss, grad = self.loss_and_grad(X_batch, y_batch)
        self.theta -= alpha * grad.reshape(-1, 1)
        pass
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
        loss_history.append(loss)
    return loss_history, self.theta
```

loss_and_grad ( ) This is the final version, which includes the consideration of m != 1, regularization, etc.

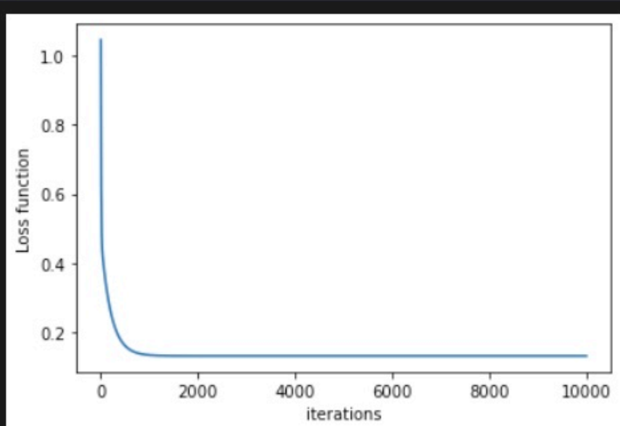
```python
def loss_and_grad(self, X, y):
    """
    Inputs:
    - X: n x d array of training data.
    - y: n x 1 targets
    Returns:
    - loss: a real number represents the loss
    - grad: a vector of the same dimensions as self.theta containing the gradient of the loss with respect to self.theta
    """
    loss = 0.0
    grad = np.zeros_like(self.theta)
    m = self.m
    n,d = X.shape
    if m==1:
        # ========================================================
        # YOUR CODE HERE:
        # Calculate the loss function of the linear regression
        # and save loss function in loss.
        # Calculate the gradient and save it as grad.
        #
        # ========================================================

        errorMatrix = self.predict(X) - y
        loss = 1 / (2 * n) * np.dot(errorMatrix.T, errorMatrix) + np.ndarray.item(self.reg / 2 * (np.dot(self.theta.T, self.theta) - pow(self.theta[0], 2)))
        grad = (1 / n * np.dot(self.get_poly_features(X).T, errorMatrix)) + (self.reg * self.theta).reshape(2, )

        # ========================================================
        # END YOUR CODE HERE
        # ========================================================
    else:
        # ======================================================== # YOUR CODE HERE:
        # Calculate the loss and gradient of the polynomial regre # with order m
        #
        # ========================================================

        errorMatrix = self.predict(X) - y
        loss = 1 / (2 * n) * np.dot(errorMatrix.T, errorMatrix) + np.ndarray.item(self.reg / 2 * (np.dot(self.theta.T, self.theta)- pow(self.theta[0], 2)))
        grad = (1 / n * np.dot(self.get_poly_features(X).T, errorMatrix)) + (self.reg * self.theta).reshape(m + 1, )

        pass
        # ========================================================
        # END YOUR CODE HERE
        # ========================================================
    return loss, grad
```

(d)(i):

gradient descent visualization:

```python
## PART (d):
## Complete train_LR function in Regression.py file
loss_history, theta = regression.train_LR(X_train,y_train, alpha=1e-2, B=30, num_iters=10000)
plt.plot(loss_history)
plt.xlabel('iterations')
plt.ylabel('Loss function')
plt.show()
print(theta)
print('Final loss:',loss_history[-1])
```

✓ 2.0s




```
[[-0.37906992]
 [ 0.8852483 ]]
Final loss: 0.13208969101982218
```

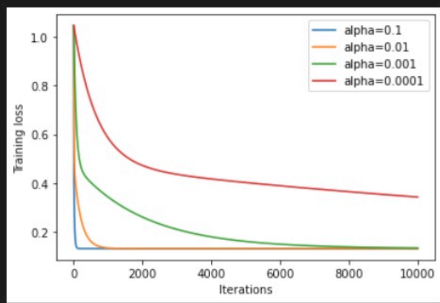
(d).(ii).

different learning rate.

We see that each line is monotonically decreasing, and bounded below by 0. So they are all convergent.

```
## PART (d) (Different Learning Rates):
from numpy.linalg import norm
alphas = [1e-1, 1e-2, 1e-3, 1e-4]
losses = np.zeros((len(alphas),10000))
# ================================================================ #
# YOUR CODE HERE:
# Train the Linear regression for different learning rates
# ================================================================ #

for i in range(0, len(alphas)):
    loss_history, theta = regression.train_LR(X_train,y_train, alpha=alphas[i], B=30, num_iters=10000)
    losses[i] = loss_history

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
fig = plt.figure()
for i, loss in enumerate(losses):
    plt.plot(range(10000), loss, label='alpha='+str(alphas[i]))
plt.xlabel('Iterations')
plt.ylabel('Training loss')
plt.legend()
plt.show()
```

✓ 7.1s

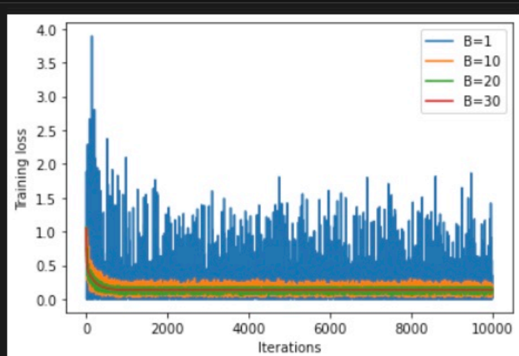


(d) (iii).

Different Batch size.

```
## PART (d) (Different Batch Sizes):
from numpy.linalg import norm
Bs = [1, 10, 20, 30]
losses = np.zeros((len(Bs),10000))
# ================================================================ #
# YOUR CODE HERE:
# Train the Linear regression for different learning rates
# ================================================================ #

for i in range(0, len(Bs)):
    loss_history, theta = regression.train_LR(X_train,y_train, alpha=1e-2, B=Bs[i], num_iters=10000)
    losses[i] = loss_history

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
fig = plt.figure()
for i, loss in enumerate(losses):
    plt.plot(range(10000), loss, label='B='+str(Bs[i]))
plt.xlabel('Iterations')
plt.ylabel('Training loss')
plt.legend()
plt.show()
fig.savefig('./LR_Batch_test.pdf')
```

✓ 5.4s



(e). closed_form()

```python
def closed_form(self, X, y):
    """
    Inputs:
    - X: n x 1 array of training data.
    - y: n x 1 array of targets
    Returns:
    - self.theta: optimal weights
    """
    m = self.m
    n,d = X.shape
    loss = 0
    if m==1:
        # ================================================================ #
        # YOUR CODE HERE:
        # obtain the optimal weights from the closed form solution
        # ================================================================ #
        polyX = self.get_poly_features(X)
        self.theta = (np.linalg.inv(polyX.T.dot(polyX))).dot(polyX.T).dot(y)
        self.theta = self.theta.reshape(-1, 1)
        loss, grad = self.loss_and_grad(X, y)
        pass
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
    else:
        # ================================================================ #
        # YOUR CODE HERE:
        # Extend X with get_poly_features().
        # Predict the targets of X.
        # ================================================================ #
        polyX = self.get_poly_features(X)
        self.theta = (np.linalg.inv(polyX.T.dot(polyX))).dot(polyX.T).dot(y)
        self.theta = self.theta.reshape(-1, 1)
        loss, grad = self.loss_and_grad(X, y)
        pass
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
    return loss, self.theta
```

The results are almost the same

```python
## PART (e):
## Complete closed_form function in Regression.py file
loss_2, theta_2 = regression.closed_form(X_train, y_train)
print('Optimal solution loss',loss_2)
print('Optimal solution theta',theta_2)
```

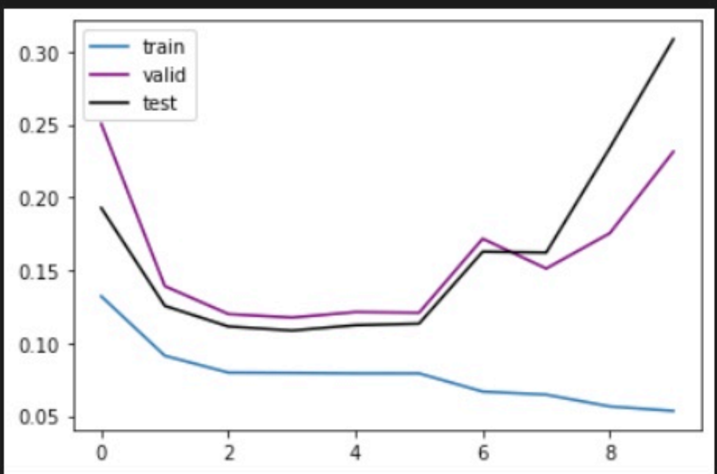
✓ 0.0s

```
Optimal solution loss 0.13208969101982218
Optimal solution theta [[-0.37906992]
 [ 0.8852483 ]]
```

(f).

```
## PART (f):
train_loss=np.zeros((10,1))
valid_loss=np.zeros((10,1))
test_loss=np.zeros((10,1))
# ================================================================ #
# YOUR CODE HERE:
# complete the following code to plot both the training, validation
# and test loss in the same plot for m range from 1 to 10
# ================================================================ #

for m in range(1, 11):
    regression = Regression(m = m)
    train_loss[m - 1] = regression.closed_form(X_train, y_train)[0]
    test_loss[m - 1] = regression.loss_and_grad(X_test, y_test)[0]
    valid_loss[m - 1] = regression.loss_and_grad(X_valid, y_valid)[0]


# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
plt.plot(train_loss, label='train')
plt.plot(valid_loss, color='purple', label='valid')
plt.plot(test_loss, color='black', label='test')
plt.legend()
plt.show()
```

✓ 0.4s



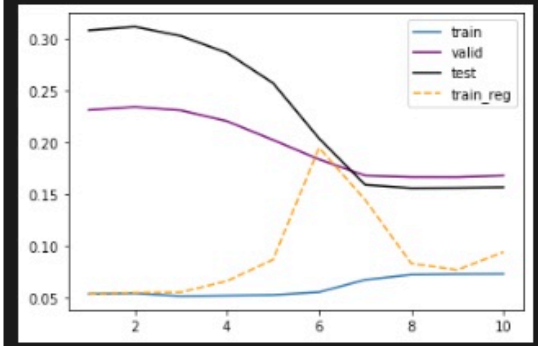
We see that the plot shows that it's underfitting when $m < 2$, and it best fits data when $2 \leqslant m \leqslant 5$, and it's overfitting when $m > 5$.

The modified get_poly_features is in part (b).

```python
#PART (g):
train_loss=np.zeros((10,1))
train_reg_loss=np.zeros((10,1))
valid_loss=np.zeros((10,1))
test_loss=np.zeros((10,1))
# ================================================================ #
# YOUR CODE HERE:
# complete the following code to plot the training, validation
# and test loss in the same plot for m range from 1 to 10
# ================================================================ #
lambdas = [0, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e0]
regression.m = 10
X_poly = regression.get_poly_features(X_train) # Assuming you have a method to get polynomial features

# ... [unchanged code before the loop]

for idx, reg in enumerate(lambdas):

    regression=Regression(10,reg)
    train_reg_loss[idx] = regression.closed_form(X_train,y_train)[0]
    train_loss[idx] = regression.loss_and_grad(X_train,y_train)[0]
    test_loss[idx] = regression.loss_and_grad(X_test,y_test)[0]
    valid_loss[idx] = regression.loss_and_grad(X_valid,y_valid)[0]

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
print(test_loss)
plt.plot(np.arange(1, 11), train_loss, label='train')
plt.plot(np.arange(1, 11), valid_loss, color='purple', label='valid')
plt.plot(np.arange(1, 11), test_loss, color='black', label='test')
plt.plot(np.arange(1, 11), train_reg_loss, color = 'orange', linestyle="dashed", label='train_reg')
plt.legend()
plt.show()
```

✓ 0.3s

```
[[0.30855252]
 [0.3120099 ]
 [0.30306526]
 [0.28667669]
 [0.25744953]
 [0.20387864]
 [0.15897954]
 [0.15579281]
 [0.15604386]
 [0.15660401]]
```



We see that $\lambda \geq 1e\text{-}3$ works better.

The modified loss_and_grad() is in part (d).

codes/Regression.py

```python
import numpy as np
import random

random.seed(10)
np.random.seed(10)

class Regression(object):
    def __init__(self, m=1, reg_param=0):
        """
        Inputs:
          - m Polynomial degree
          - regularization parameter reg_param
        Goal:
         - Initialize the weight vector self.theta
         - Initialize the polynomial degree self.m
         - Initialize the  regularization parameter self.reg
        """
        self.m = m
        self.reg  = reg_param
        self.dim = [m+1 , 1]
        ### These two lines set the random seeds... you can ignore. #####
        random.seed(10)
        np.random.seed(10)
        #################################################################
        self.theta = np.random.standard_normal(self.dim)
    def get_poly_features(self, X):
        """
        Inputs:
         - X: A numpy array of shape (n,1) containing the data.
        Returns:
         - X_out: an augmented training data as an mth degree feature vector
         e.g. [1, x, x^2, ..., x^m], x \in X.
        """
        n,d = X.shape
        m = self.m
        X_out= np.zeros((n,m+1))
        if m==1:
            # ================================================================ #
            # YOUR CODE HERE:
            # IMPLEMENT THE MATRIX X_out with each entry = [1, x]
            # ================================================================ #
            for i in range(0, n):
                X_out[i, :] = np.array([1, X[i, 0]])
            pass
            # ================================================================ #
            # END YOUR CODE HERE
            # ================================================================ #
        else:
            # ================================================================ #
            # YOUR CODE HERE:
            # IMPLEMENT THE MATRIX X_out with each entry = [1, x, x^2,....,x^m]
```

```
            # ================================================================ #
            for i in range(0, n):
                for j in range(0, m+1):
                    X_out[i, j] = pow(X[i, 0], j)
            pass
            # ================================================================ #
            # END YOUR CODE HERE
            # ================================================================ #
            pass
        return X_out

    def loss_and_grad(self, X, y):
        """
        Inputs:
        - X: n x d array of training data.
        - y: n x 1 targets
        Returns:
        - loss: a real number represents the loss
        - grad: a vector of the same dimensions as self.theta containing the gradient of the loss with respect to self.theta
        """
        loss = 0.0
        grad = np.zeros_like(self.theta)
        m = self.m
        n,d = X.shape
        if m==1:
            # ========================================================
            # YOUR CODE HERE:
            # Calculate the loss function of the linear regression
            # and save loss function in loss.
            # Calculate the gradient and save it as grad.
            #
            # ========================================================

            errorMatrix = self.predict(X) - y
            loss = 1 / (2 * n) * np.dot(errorMatrix.T, errorMatrix) + np.ndarray.item(self.reg / 2 * (np.dot(self.theta.T, self.theta) - pow(self.theta[0], 2)))
            grad = (1 / n * np.dot(self.get_poly_features(X).T, errorMatrix)) + (self.reg * self.theta).reshape(2, )

            # ========================================================
            # END YOUR CODE HERE
            # ========================================================
        else:
            # ======================================================== # YOUR CODE HERE:
            # Calculate the loss and gradient of the polynomial regre # with order m
            #
            # ========================================================

            errorMatrix = self.predict(X) - y
            loss = 1 / (2 * n) * np.dot(errorMatrix.T, errorMatrix) + np.ndarray.item(self.reg / 2 * (np.dot(self.theta.T, self.theta)-
```

```
pow(self.theta[0], 2)))
            grad = (1 / n * np.dot(self.get_poly_features(X).T, errorMatrix)) + 
(self.reg * self.theta).reshape(m + 1, )

            pass
            # ================================================================
            # END YOUR CODE HERE
            # ================================================================
        return loss, grad

    def train_LR(self, X, y, alpha=1e-2, B=30, num_iters=10000) :
        """
        Finds the coefficients of a {d-1}^th degree polynomial
        that fits the data using least squares mini-batch gradient descent.

        Inputs:
         - X         -- numpy array of shape (n,d), features
         - y         -- numpy array of shape (n,), targets
         - alpha     -- float, learning rate
         -B          -- integer, batch size
         - num_iters -- integer, maximum number of iterations

        Returns:
         - loss_history: vector containing the loss at each training iteration.
         - self.theta: optimal weights
        """
        ### These two lines set the random seeds... you can ignore. #####
        random.seed(10)
        np.random.seed(10)
        #################################################################
        self.theta = np.random.standard_normal(self.dim)
        loss_history = []
        n,d = X.shape
        for t in np.arange(num_iters):
            X_batch = None
            y_batch = None
            # ================================================================ #
            # YOUR CODE HERE:
            # Shuffle X, y along the batch axis with np.random.shuffle.
            # Get the first batch_size elements X_batch from X, y_batch from Y.
            # X_batch should have shape: (B,1), y_batch should have shape: (B,).
            # ================================================================ #
            indices = np.arange(n)
            np.random.shuffle(indices)
            X = X[indices]
            y = y[indices]
            X_batch = X[0:B]
            y_batch = y[0:B]
            pass
            # ================================================================ #
            # END YOUR CODE HERE
            # ================================================================ #
            loss = 0.0
            grad = np.zeros_like(self.theta)
```

```
            # ================================================================ #
            # YOUR CODE HERE:
            # evaluate loss and gradient for batch data
            # save loss as loss and gradient as grad
            # update the weights self.theta
            # ================================================================ #
            loss, grad = self.loss_and_grad(X_batch, y_batch)
            self.theta -= alpha * grad.reshape(-1, 1)
            pass
            # ================================================================ #
            # END YOUR CODE HERE
            # ================================================================ #
            loss_history.append(loss)
        return loss_history, self.theta



    def closed_form(self, X, y):
        """
        Inputs:
        - X: n x 1 array of training data.
        - y: n x 1 array of targets
        Returns:
        - self.theta: optimal weights
        """
        m = self.m
        n,d = X.shape
        loss = 0
        if m==1:
            # ================================================================ #
            # YOUR CODE HERE:
            # obtain the optimal weights from the closed form solution
            # ================================================================ #
            polyX = self.get_poly_features(X)
            self.theta = (np.linalg.inv(polyX.T.dot(polyX))).dot(polyX.T).dot(y)
            self.theta = self.theta.reshape(-1, 1)
            loss, grad = self.loss_and_grad(X, y)
            pass
            # ================================================================ #
            # END YOUR CODE HERE
            # ================================================================ #
        else:
            # ================================================================ #
            # YOUR CODE HERE:
            # Extend X with get_poly_features().
            # Predict the targets of X.
            # ================================================================ #
            polyX = self.get_poly_features(X)
            self.theta = (np.linalg.inv(polyX.T.dot(polyX))).dot(polyX.T).dot(y)
            self.theta = self.theta.reshape(-1, 1)
            loss, grad = self.loss_and_grad(X, y)
            pass
            # ================================================================ #
            # END YOUR CODE HERE
```

```python
            # ================================================================ #
        return loss, self.theta


    def predict(self, X):
        """
        Inputs:
        - X: n x 1 array of training data.
        Returns:
        - y_pred: Predicted targets for the data in X. y_pred is a 1-dimensional
          array of length n.
        """
        y_pred = np.zeros(X.shape[0])
        m = self.m
        theta = self.theta
        if m==1:
            # ================================================================ #
            # YOUR CODE HERE:
            # PREDICT THE TARGETS OF X
            # ================================================================ #
            for i in range(0, X.shape[0]):
                y_pred[i] = theta[0] + theta[1] * X[i]
            pass
            # ================================================================ #
            # END YOUR CODE HERE
            # ================================================================ #
        else:
            # ================================================================ #
            # YOUR CODE HERE:
            # Extend X with get_poly_features().
            # Predict the target of X.
            # ================================================================ #
            polyX = self.get_poly_features(X)
            for i in range(0, polyX.shape[0]):
                for j in range(0, polyX.shape[1]):
                    y_pred[i] += theta[j] * polyX[i, j]
            pass
            # ================================================================ #
            # END YOUR CODE HERE
            # ================================================================ #
        return y_pred
```

Notebook.py

```
# %%
import numpy as np
import matplotlib.pyplot as plt
import random
import csv
from utils.data_load import load
import codes
# Load matplotlib images inline
%matplotlib inline
# These are important for reloading any code you write in external .py files.
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

# %% [markdown]
# # Problem 4: Linear Regression
# Please follow our instructions in the same order to solve the linear regresssion problem.
#
# Please print out the entire results and codes when completed.

# %%
def get_data():
    """
    Load the dataset from disk and perform preprocessing to prepare it for the linear regression problem.
    """
    X_train, y_train = load('./data/regression/regression_train.csv')
    X_test, y_test = load('./data/regression/regression_test.csv')
    X_valid, y_valid = load('./data/regression/regression_valid.csv')
    return X_train, y_train, X_test, y_test, X_valid, y_valid

X_train, y_train, X_test, y_test, X_valid, y_valid= get_data()


print('Train data shape: ', X_train.shape)
print('Train target shape: ', y_train.shape)
print('Test data shape: ',X_test.shape)
print('Test target shape: ',y_test.shape)
print('Valid data shape: ',X_valid.shape)
print('Valid target shape: ',y_valid.shape)

# %%
## PART (a):
## Plot the training and test data ##

plt.plot(X_train, y_train,'o', color='black')
plt.plot(X_test, y_test,'o', color='blue')
plt.xlabel('Input')
plt.ylabel('Target')
plt.show()
```

```python

# %% [markdown]
# ## Training Linear Regression
# In the following cells, you will build a linear regression. You will implement its loss function, then subsequently train it with gradient descent. You will choose the learning rate of gradient descent to optimize its classification performance. Finally, you will get the opimal solution using closed form expression.

# %%
from codes.Regression import Regression

# %%
## PART (c):
## Complete loss_and_grad function in Regression.py file and test your results.
regression = Regression(m=1, reg_param=0)
loss, grad = regression.loss_and_grad(X_train,y_train)
print('Loss value',loss)
print('Gradient value',grad)

##


# %%
## PART (d):
## Complete train_LR function in Regression.py file
loss_history, theta = regression.train_LR(X_train,y_train, alpha=1e-2, B=30, num_iters=10000)
plt.plot(loss_history)
plt.xlabel('iterations')
plt.ylabel('Loss function')
plt.show()
print(theta)
print('Final loss:',loss_history[-1])

# %%
## PART (d) (Different Learning Rates):
from numpy.linalg import norm
alphas = [1e-1, 1e-2, 1e-3, 1e-4]
losses = np.zeros((len(alphas),10000))
# ================================================================ #
# YOUR CODE HERE:
# Train the Linear regression for different learning rates
# ================================================================ #

for i in range(0, len(alphas)):
    loss_history, theta = regression.train_LR(X_train,y_train, alpha=alphas[i], B=30, num_iters=10000)
    losses[i] = loss_history

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
fig = plt.figure()
for i, loss in enumerate(losses):
    plt.plot(range(10000), loss, label='alpha='+str(alphas[i]))
```

```python
plt.xlabel('Iterations')
plt.ylabel('Training loss')
plt.legend()
plt.show()

# %%
## PART (d) (Different Batch Sizes):
from numpy.linalg import norm
Bs = [1, 10, 20, 30]
losses = np.zeros((len(Bs),10000))
# ================================================================ #
# YOUR CODE HERE:
# Train the Linear regression for different learning rates
# ================================================================ #

for i in range(0, len(Bs)):
    loss_history, theta = regression.train_LR(X_train,y_train, alpha=1e-2, B=Bs[i], num_iters=10000)
    losses[i] = loss_history

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
fig = plt.figure()
for i, loss in enumerate(losses):
    plt.plot(range(10000), loss, label='B='+str(Bs[i]))
plt.xlabel('Iterations')
plt.ylabel('Training loss')
plt.legend()
plt.show()
fig.savefig('./LR_Batch_test.pdf')

# %%
## PART (e):
## Complete closed_form function in Regression.py file
loss_2, theta_2 = regression.closed_form(X_train, y_train)
print('Optimal solution loss',loss_2)
print('Optimal solution theta',theta_2)

# %%
## PART (f):
train_loss=np.zeros((10,1))
valid_loss=np.zeros((10,1))
test_loss=np.zeros((10,1))
# ================================================================ #
# YOUR CODE HERE:
# complete the following code to plot both the training, validation
# and test loss in the same plot for m range from 1 to 10
# ================================================================ #

for m in range(1, 11):
    regression = Regression(m = m)
    train_loss[m - 1] = regression.closed_form(X_train, y_train)[0]
    test_loss[m - 1] = regression.loss_and_grad(X_test, y_test)[0]
```

```
    valid_loss[m - 1] = regression.loss_and_grad(X_valid, y_valid)[0]


# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
plt.plot(train_loss, label='train')
plt.plot(valid_loss, color='purple', label='valid')
plt.plot(test_loss, color='black', label='test')
plt.legend()
plt.show()

# %%
#PART (g):
train_loss=np.zeros((10,1))
train_reg_loss=np.zeros((10,1))
valid_loss=np.zeros((10,1))
test_loss=np.zeros((10,1))
# ================================================================ #
# YOUR CODE HERE:
# complete the following code to plot the training, validation
# and test loss in the same plot for m range from 1 to 10
# ================================================================ #
lambdas = [0, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e0]
regression.m = 10
X_poly = regression.get_poly_features(X_train) # Assuming you have a method to get polynomial features

# ... [unchanged code before the loop]

for idx, reg in enumerate(lambdas):

    regression=Regression(10,reg)
    train_reg_loss[idx] = regression.closed_form(X_train,y_train)[0]
    train_loss[idx] = regression.loss_and_grad(X_train,y_train)[0]
    test_loss[idx] = regression.loss_and_grad(X_test,y_test)[0]
    valid_loss[idx] = regression.loss_and_grad(X_valid,y_valid)[0]

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
print(test_loss)
plt.plot(np.arange(1, 11), train_loss, label='train')
plt.plot(np.arange(1, 11), valid_loss, color='purple', label='valid')
plt.plot(np.arange(1, 11), test_loss, color='black', label='test')
plt.plot(np.arange(1, 11), train_reg_loss, color = 'orange', linestyle="dashed", label='train_reg')
plt.legend()
plt.show()
```