

Problem 1:

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

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

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

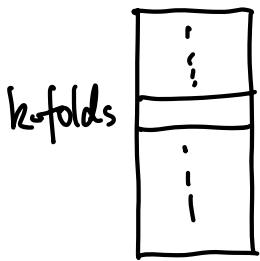
$$\Rightarrow \frac{1}{2n} (2X^T X\theta - 2X^T y) + \lambda\theta = 0$$

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

$$\theta = \left(X^T X + n\lambda \cdot \overset{\uparrow}{I_{d+1}} \right)^{-1} X^T y$$

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

(b) True.



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

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

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

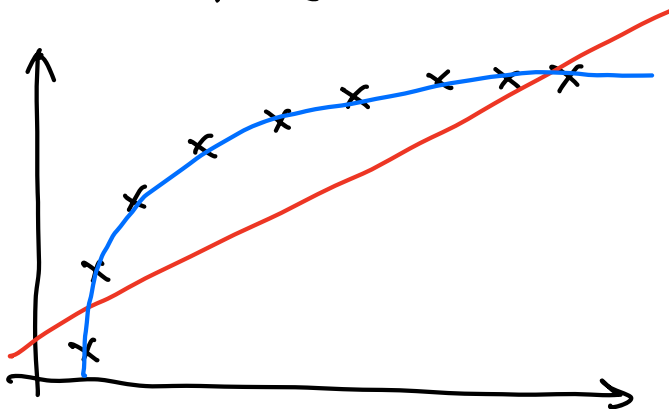
$$\frac{\partial}{\partial \theta} (\phi(x) \theta^T \theta \phi(x)^T - 2 y \theta \phi(x)^T - y y^T) = 0$$

$$\Rightarrow \phi(x)^T \phi(x) \theta - \phi(x)^T y = 0$$

$$\Rightarrow \theta = (\phi(x)^T \phi(x))^{-1} \phi(x)^T y$$

(d). A.

The relationship may be non-linear. For example:



We see that the red line is a linear relationship, but the true relationship is non-linear (blue curve).

(e). C.

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

Problem 2:

(a)

$$\begin{aligned}
 J_{MAE}(\theta) &= \frac{1}{n} \sum_{i=1}^n |\theta^T x^{(i)} - y^{(i)}| \\
 &= \frac{1}{n} \sum_{i=1}^n \text{sign}(\theta^T x^{(i)} - y^{(i)}) \cdot (\theta^T x^{(i)} - y^{(i)}) \\
 \text{so } \frac{\partial}{\partial \theta_j} J_{MAE}(\theta) &= \frac{\partial}{\partial \theta_j} \left(\frac{1}{n} \sum_{i=1}^n \text{sign}(\theta^T x^{(i)} - y^{(i)}) \cdot (\theta^T x^{(i)} - y^{(i)}) \right) \\
 &= \frac{1}{n} \sum_{i=1}^n \text{sign}(\theta^T x^{(i)} - y^{(i)}) \cdot \frac{\partial}{\partial \theta_j} (\theta^T x^{(i)}) \\
 &= \frac{1}{n} \sum_{i=1}^n \text{sign}(\theta^T x^{(i)} - y^{(i)}) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{k=1}^d \theta_k x_k^{(i)} - y^{(i)} \right) \\
 &= \frac{1}{n} \sum_{i=1}^n \text{sign}(\theta^T x^{(i)} - y^{(i)}) \cdot x_j^{(i)}
 \end{aligned}$$

(b).

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

(c).

$$\begin{aligned}
 J_{MAE}(\theta) &= \frac{1}{n} \sum_{i=1}^n |\theta^T x^{(i)} - y^{(i)}| \\
 &= \frac{1}{n} \cdot \sum_{i=1}^n \left| \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}^T x^{(i)} - y^{(i)} \right| \\
 &= \frac{1}{n} \cdot \sum_{i=1}^n |x_0^{(i)} + x_1^{(i)} + x_2^{(i)} - y^{(i)}| \\
 &= \frac{1}{8} \cdot ((12-1-4) + (3-1-1-1) + (1+1-1) + (6-1+2+2) + (3-1+2-1) + (6-1-1) + (8-5-2-1) + (7-3-1)) \\
 &= \frac{1}{8} \cdot (7 + 1 + 9 + 3 + 4 + 3) \\
 &= \frac{27}{8}
 \end{aligned}$$

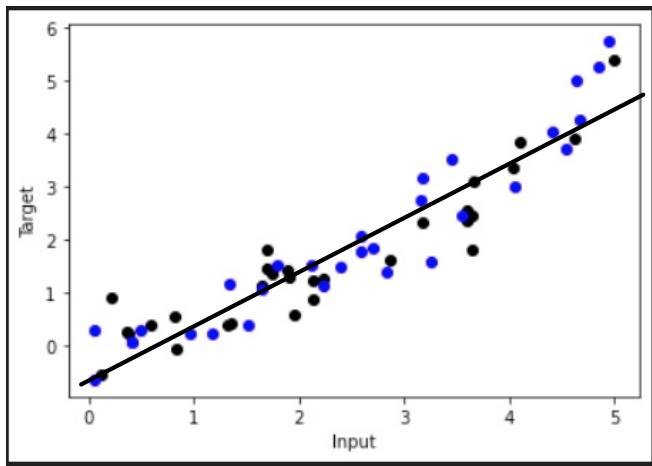
$$\begin{aligned}
J_{MSE}(\theta) &= \frac{1}{n} \sum_{i=1}^n (h_{\theta}(x^{(i)}) - y^{(i)})^2 \\
&= \frac{1}{n} \sum_{i=1}^n (\theta^T x^{(i)} - y^{(i)})^2 \\
&= \frac{1}{n} \sum_{i=1}^n (x_0^{(i)} + x_1^{(i)} + x_2^{(i)} - y^{(i)})^2 \\
&= \frac{1}{8} ((12-1-4)^2 + (3-1-1-1)^2 + (1+1-1)^2 + (6-1+2+2)^2 + (3-1+2-1)^2 + (6-1-1)^2 + (8-5-2-1)^2 + (7-3-1)^2) \\
&= \frac{1}{8} (7^2 + 1^2 + 0^2 + 3^2 + 4^2 + 3) \\
&= \frac{1}{8} (49 + 1 + 0 + 9 + 16 + 9) \\
&= \frac{1}{8} \cdot 165 \\
&= \frac{165}{8}
\end{aligned}$$

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

$$\begin{aligned}
\nabla_{\theta} J_{MSE}(\theta) &= \nabla_{\theta} \frac{1}{n} (X\theta - y)^T (X\theta - y) \\
&= \frac{1}{n} \nabla_{\theta} (\theta^T X^T X \theta - 2\theta^T X^T y + y^T y) \\
&= \frac{1}{n} \nabla_{\theta} (\text{tr} \theta^T X^T X \theta - 2 \text{tr} \theta^T X^T y) \\
&= \frac{2}{n} (X^T X \theta - X^T y) \\
&= \frac{2}{8} \left(\begin{bmatrix} 21 \\ 63 \\ 27 \end{bmatrix} - \begin{bmatrix} 46 \\ 58 \\ 11 \end{bmatrix} \right) \\
&= \frac{2}{8} \begin{bmatrix} -25 \\ 25 \\ 16 \end{bmatrix} = \begin{bmatrix} -6.25 \\ 6.25 \\ 4 \end{bmatrix}
\end{aligned}$$

Problem 3:

(a)



From the plot, we see that it's mostly linear, 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()

```
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:

```
## 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()`.

```
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 $m \neq 1$, regularization, etc.

```
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, )

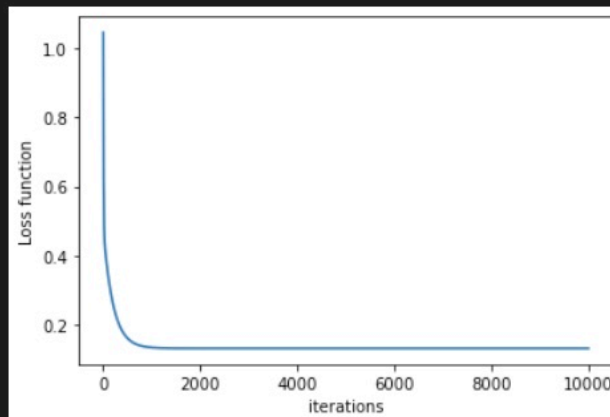
        # =====
        # END YOUR CODE HERE
        # =====
    else:
        # ===== # YOUR CODE HERE:
        # Calculate the loss and gradient of the polynomial regre # with order m
        # =====
        errorMatrix = self.predict(X) - y
        loss = 1 / (2 * n) * np.dot(errorMatrix.T, errorMatrix) + 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:

```
## 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.

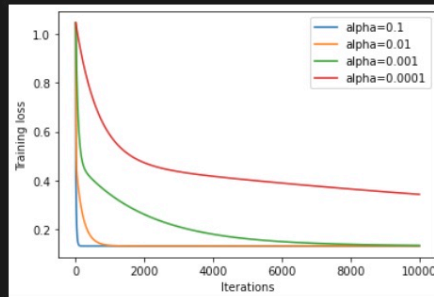
We see that each line is monotonically decreasing, and bounded below by 0. So they are all convergent.

```
## 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).

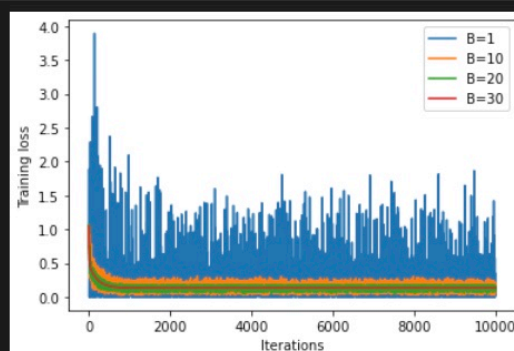
Different Batch size.

```
## 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()

```
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

```
## 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 ]]
```

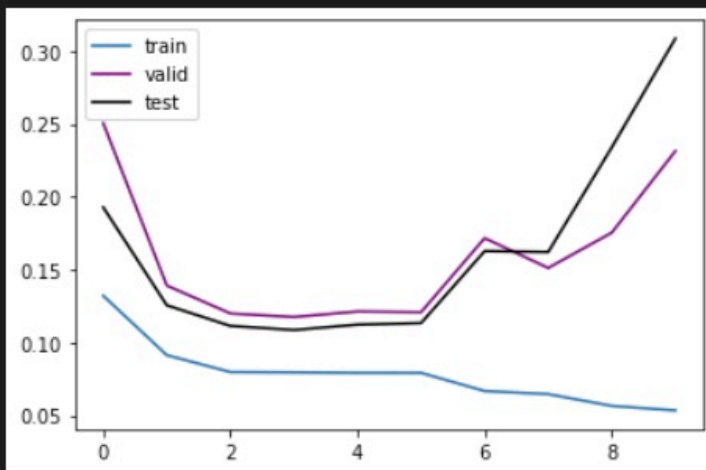
(f).

```
## 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 l^2 's underfitting when $m < 2$, and it best fits data when $2 \leq m \leq 5$, and it's overfitting when $m > 5$.

The modified get_poly_features is in part (b).

```

#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()

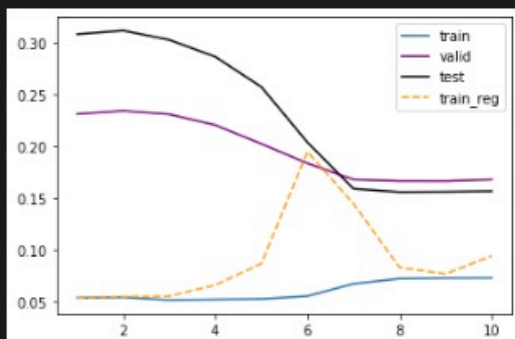
```

✓ 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 $\lambda \geq 1e-3$ works better.

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

codes/Regression.py

```

1  import numpy as np
2  import random
3
4  random.seed(10)
5  np.random.seed(10)
6
7  class Regression(object):
8      def __init__(self, m=1, reg_param=0):
9          """
10         Inputs:
11             - m Polynomial degree
12             - regularization parameter reg_param
13         Goal:
14             - Initialize the weight vector self.theta
15             - Initialize the polynomial degree self.m
16             - Initialize the regularization parameter self.reg
17         """
18         self.m = m
19         self.reg = reg_param
20         self.dim = [m+1, 1]
21         ### These two lines set the random seeds... you can ignore. #####
22         random.seed(10)
23         np.random.seed(10)
24         #####
25         self.theta = np.random.standard_normal(self.dim)
26     def get_poly_features(self, X):
27         """
28         Inputs:
29             - X: A numpy array of shape (n,1) containing the data.
30         Returns:
31             - X_out: an augmented training data as an mth degree feature vector
32               e.g. [1, x, x^2, ..., x^m], x \in X.
33         """
34         n,d = X.shape
35         m = self.m
36         X_out= np.zeros((n,m+1))
37         if m==1:
38             # ===== #
39             # YOUR CODE HERE:
40             # IMPLEMENT THE MATRIX X_out with each entry = [1, x]
41             # ===== #
42             for i in range(0, n):
43                 X_out[i, :] = np.array([1, X[i, 0]])
44             pass
45             # ===== #
46             # END YOUR CODE HERE
47             # ===== #
48         else:
49             # ===== #
50             # YOUR CODE HERE:
51             # IMPLEMENT THE MATRIX X_out with each entry = [1, x, x^2, ..., x^m]

```



```

52         # ===== #
53         for i in range(0, n):
54             for j in range(0, m+1):
55                 X_out[i, j] = pow(X[i, 0], j)
56         pass
57         # ===== #
58         # END YOUR CODE HERE
59         # ===== #
60         pass
61     return X_out
62
63     def loss_and_grad(self, X, y):
64         """
65         Inputs:
66         - X: n x d array of training data.
67         - y: n x 1 targets
68         Returns:
69         - loss: a real number represents the loss
70         - grad: a vector of the same dimensions as self.theta containing the
71           gradient of the loss with respect to self.theta
72         """
73         loss = 0.0
74         grad = np.zeros_like(self.theta)
75         m = self.m
76         n, d = X.shape
77         if m==1:
78             # =====
79             # YOUR CODE HERE:
80             # Calculate the loss function of the linear regression
81             # and save loss function in loss.
82             # Calculate the gradient and save it as grad.
83             # =====
84
85             errorMatrix = self.predict(X) - y
86             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)))
87             grad = (1 / n * np.dot(self.get_poly_features(X).T, errorMatrix)) +
(self.reg * self.theta).reshape(2, )
88
89             # =====
90             # END YOUR CODE HERE
91             # =====
92         else:
93             # ===== # YOUR CODE
94             HERE:
95             # Calculate the loss and gradient of the polynomial regre # with order
96             m
97             #
98             # =====
99
100             errorMatrix = self.predict(X) - y
101             loss = 1 / (2 * n) * np.dot(errorMatrix.T, errorMatrix) +
np.ndarray.item(self.reg / 2 * (np.dot(self.theta.T, self.theta)-

```

```

100     pow(self.theta[0], 2)))
101     grad = (1 / n * np.dot(self.get_poly_features(X).T, errorMatrix)) +
102     (self.reg * self.theta).reshape(m + 1, )
103     pass
104     # =====
105     # END YOUR CODE HERE
106     # =====
107     return loss, grad
108
109 def train_LR(self, X, y, alpha=1e-2, B=30, num_iters=10000) :
110     """
111     Finds the coefficients of a {d-1}^th degree polynomial
112     that fits the data using least squares mini-batch gradient descent.
113
114     Inputs:
115     - X          -- numpy array of shape (n,d), features
116     - y          -- numpy array of shape (n,), targets
117     - alpha      -- float, learning rate
118     - B          -- integer, batch size
119     - num_iters  -- integer, maximum number of iterations
120
121     Returns:
122     - loss_history: vector containing the loss at each training iteration.
123     - self.theta: optimal weights
124     """
125     ### These two lines set the random seeds... you can ignore. #####
126     random.seed(10)
127     np.random.seed(10)
128     #####
129     self.theta = np.random.standard_normal(self.dim)
130     loss_history = []
131     n,d = X.shape
132     for t in np.arange(num_iters):
133         X_batch = None
134         y_batch = None
135         # ===== #
136         # YOUR CODE HERE:
137         # Shuffle X, y along the batch axis with np.random.shuffle.
138         # Get the first batch_size elements X_batch from X, y_batch from Y.
139         # X_batch should have shape: (B,1), y_batch should have shape: (B,).
140         # ===== #
141         indices = np.arange(n)
142         np.random.shuffle(indices)
143         X = X[indices]
144         y = y[indices]
145         X_batch = X[0:B]
146         y_batch = y[0:B]
147         pass
148         # ===== #
149         # END YOUR CODE HERE
150         # ===== #
151         loss = 0.0
152         grad = np.zeros_like(self.theta)

```



```

152         # ===== #
153         # YOUR CODE HERE:
154         # evaluate loss and gradient for batch data
155         # save loss as loss and gradient as grad
156         # update the weights self.theta
157         # ===== #
158         loss, grad = self.loss_and_grad(X_batch, y_batch)
159         self.theta -= alpha * grad.reshape(-1, 1)
160         pass
161         # ===== #
162         # END YOUR CODE HERE
163         # ===== #
164         loss_history.append(loss)
165     return loss_history, self.theta
166
167
168
169     def closed_form(self, X, y):
170         """
171         Inputs:
172         - X: n x 1 array of training data.
173         - y: n x 1 array of targets
174         Returns:
175         - self.theta: optimal weights
176         """
177         m = self.m
178         n,d = X.shape
179         loss = 0
180         if m==1:
181             # ===== #
182             # YOUR CODE HERE:
183             # obtain the optimal weights from the closed form solution
184             # ===== #
185             polyX = self.get_poly_features(X)
186             self.theta = (np.linalg.inv(polyX.T.dot(polyX))).dot(polyX.T).dot(y)
187             self.theta = self.theta.reshape(-1, 1)
188             loss, grad = self.loss_and_grad(X, y)
189             pass
190             # ===== #
191             # END YOUR CODE HERE
192             # ===== #
193         else:
194             # ===== #
195             # YOUR CODE HERE:
196             # Extend X with get_poly_features().
197             # Predict the targets of X.
198             # ===== #
199             polyX = self.get_poly_features(X)
200             self.theta = (np.linalg.inv(polyX.T.dot(polyX))).dot(polyX.T).dot(y)
201             self.theta = self.theta.reshape(-1, 1)
202             loss, grad = self.loss_and_grad(X, y)
203             pass
204             # ===== #
205             # END YOUR CODE HERE

```

```
206         # ===== #
207     return loss, self.theta
208
209
210     def predict(self, X):
211         """
212         Inputs:
213         - X: n x 1 array of training data.
214         Returns:
215         - y_pred: Predicted targets for the data in X. y_pred is a 1-dimensional
216           array of length n.
217         """
218         y_pred = np.zeros(X.shape[0])
219         m = self.m
220         theta = self.theta
221         if m==1:
222             # ===== #
223             # YOUR CODE HERE:
224             # PREDICT THE TARGETS OF X
225             # ===== #
226             for i in range(0, X.shape[0]):
227                 y_pred[i] = theta[0] + theta[1] * X[i]
228             pass
229             # ===== #
230             # END YOUR CODE HERE
231             # ===== #
232         else:
233             # ===== #
234             # YOUR CODE HERE:
235             # Extend X with get_poly_features().
236             # Predict the target of X.
237             # ===== #
238             polyX = self.get_poly_features(X)
239             for i in range(0, polyX.shape[0]):
240                 for j in range(0, polyX.shape[1]):
241                     y_pred[i] += theta[j] * polyX[i, j]
242             pass
243             # ===== #
244             # END YOUR CODE HERE
245             # ===== #
246         return y_pred
```

Notebook.py

```
1  # %%
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import random
5  import csv
6  from utils.data_load import load
7  import codes
8  # Load matplotlib images inline
9  %matplotlib inline
10 # These are important for reloading any code you write in external .py files.
11 # see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
12 %load_ext autoreload
13 %autoreload 2
14
15 # %% [markdown]
16 # # Problem 4: Linear Regression
17 # Please follow our instructions in the same order to solve the linear regresssion
18 # problem.
19 # Please print out the entire results and codes when completed.
20
21 # %%
22 def get_data():
23     """
24     Load the dataset from disk and perform preprocessing to prepare it for the
25     linear regression problem.
26     """
27     X_train, y_train = load('./data/regression/regression_train.csv')
28     X_test, y_test = load('./data/regression/regression_test.csv')
29     X_valid, y_valid = load('./data/regression/regression_valid.csv')
30     return X_train, y_train, X_test, y_test, X_valid, y_valid
31
32 X_train, y_train, X_test, y_test, X_valid, y_valid= get_data()
33
34 print('Train data shape: ', X_train.shape)
35 print('Train target shape: ', y_train.shape)
36 print('Test data shape: ', X_test.shape)
37 print('Test target shape: ', y_test.shape)
38 print('Valid data shape: ', X_valid.shape)
39 print('Valid target shape: ', y_valid.shape)
40
41 # %%
42 ## PART (a):
43 ## Plot the training and test data ##
44
45 plt.plot(X_train, y_train, 'o', color='black')
46 plt.plot(X_test, y_test, 'o', color='blue')
47 plt.xlabel('Input')
48 plt.ylabel('Target')
49 plt.show()
```

```

50
51 # %% [markdown]
52 # ## Training Linear Regression
53 # 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.
54
55 # %%
56 from codes.Regression import Regression
57
58 # %%
59 ## PART (c):
60 ## Complete loss_and_grad function in Regression.py file and test your results.
61 regression = Regression(m=1, reg_param=0)
62 loss, grad = regression.loss_and_grad(X_train,y_train)
63 print('Loss value',loss)
64 print('Gradient value',grad)
65
66 ##
67
68 # %%
69 ## PART (d):
70 ## Complete train_LR function in Regression.py file
71 loss_history, theta = regression.train_LR(X_train,y_train, alpha=1e-2, B=30,
  num_iters=10000)
72 plt.plot(loss_history)
73 plt.xlabel('iterations')
74 plt.ylabel('Loss function')
75 plt.show()
76 print(theta)
77 print('Final loss:',loss_history[-1])
78
79 # %%
80 ## PART (d) (Different Learning Rates):
81 from numpy.linalg import norm
82 alphas = [1e-1, 1e-2, 1e-3, 1e-4]
83 losses = np.zeros((len(alphas),10000))
84 # ===== #
85 # YOUR CODE HERE:
86 # Train the Linear regression for different learning rates
87 # ===== #
88
89 for i in range(0, len(alphas)):
90     loss_history, theta = regression.train_LR(X_train,y_train, alpha=alphas[i], B=
  30, num_iters=10000)
91     losses[i] = loss_history
92
93 # ===== #
94 # END YOUR CODE HERE
95 # ===== #
96 fig = plt.figure()
97 for i, loss in enumerate(losses):
98     plt.plot(range(10000), loss, label='alpha='+str(alphas[i]))

```

```

99 plt.xlabel('Iterations')
100 plt.ylabel('Training loss')
101 plt.legend()
102 plt.show()
103
104 # %%
105 ## PART (d) (Different Batch Sizes):
106 from numpy.linalg import norm
107 Bs = [1, 10, 20, 30]
108 losses = np.zeros((len(Bs),10000))
109 # ===== #
110 # YOUR CODE HERE:
111 # Train the Linear regression for different learning rates
112 # ===== #
113
114 for i in range(0, len(Bs)):
115     loss_history, theta = regression.train_LR(X_train,y_train, alpha=1e-2, B=Bs[i]
116     , num_iters=10000)
117     losses[i] = loss_history
118 # ===== #
119 # END YOUR CODE HERE
120 # ===== #
121 fig = plt.figure()
122 for i, loss in enumerate(losses):
123     plt.plot(range(10000), loss, label='B='+str(Bs[i]))
124 plt.xlabel('Iterations')
125 plt.ylabel('Training loss')
126 plt.legend()
127 plt.show()
128 fig.savefig('./LR_Batch_test.pdf')
129
130 # %%
131 ## PART (e):
132 ## Complete closed_form function in Regression.py file
133 loss_2, theta_2 = regression.closed_form(X_train, y_train)
134 print('Optimal solution loss',loss_2)
135 print('Optimal solution theta',theta_2)
136
137 # %%
138 ## PART (f):
139 train_loss=np.zeros((10,1))
140 valid_loss=np.zeros((10,1))
141 test_loss=np.zeros((10,1))
142 # ===== #
143 # YOUR CODE HERE:
144 # complete the following code to plot both the training, validation
145 # and test loss in the same plot for m range from 1 to 10
146 # ===== #
147
148 for m in range(1, 11):
149     regression = Regression(m = m)
150     train_loss[m - 1] = regression.closed_form(X_train, y_train)[0]
151     test_loss[m - 1] = regression.loss_and_grad(X_test, y_test)[0]

```

```

152     valid_loss[m - 1] = regression.loss_and_grad(X_valid, y_valid)[0]
153
154
155 # ===== #
156 # END YOUR CODE HERE
157 # ===== #
158 plt.plot(train_loss, label='train')
159 plt.plot(valid_loss, color='purple', label='valid')
160 plt.plot(test_loss, color='black', label='test')
161 plt.legend()
162 plt.show()
163
164 # %%
165 #PART (g):
166 train_loss=np.zeros((10,1))
167 train_reg_loss=np.zeros((10,1))
168 valid_loss=np.zeros((10,1))
169 test_loss=np.zeros((10,1))
170 # ===== #
171 # YOUR CODE HERE:
172 # complete the following code to plot the training, validation
173 # and test loss in the same plot for m range from 1 to 10
174 # ===== #
175 lambdas = [0, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e0]
176 regression.m = 10
177 X_poly = regression.get_poly_features(X_train) # Assuming you have a method to get
178 polynomial features
179 # ... [unchanged code before the loop]
180
181 for idx, reg in enumerate(lambdas):
182
183     regression=Regression(10,reg)
184     train_reg_loss[idx] = regression.closed_form(X_train,y_train)[0]
185     train_loss[idx] = regression.loss_and_grad(X_train,y_train)[0]
186     test_loss[idx] = regression.loss_and_grad(X_test,y_test)[0]
187     valid_loss[idx] = regression.loss_and_grad(X_valid,y_valid)[0]
188
189 # ===== #
190 # END YOUR CODE HERE
191 # ===== #
192 print(test_loss)
193 plt.plot(np.arange(1, 11), train_loss, label='train')
194 plt.plot(np.arange(1, 11), valid_loss, color='purple', label='valid')
195 plt.plot(np.arange(1, 11), test_loss, color='black', label='test')
196 plt.plot(np.arange(1, 11), train_reg_loss, color = 'orange', linestyle="dashed",
197 label='train_reg')
198 plt.legend()
199 plt.show()
200
201

```