Homework 1 **KRISH PATEL** **605796227**

CS M146

Problem 1:

a) False: Ridge Regression has a closed form solution. Ridge regression uses L2 Norm regularization to help overcome overfitting and underfitting. The reason it is closed form is due to the fact that the loss function is convex and is differentiable. It can be expressed in a finite number of mathematic operations

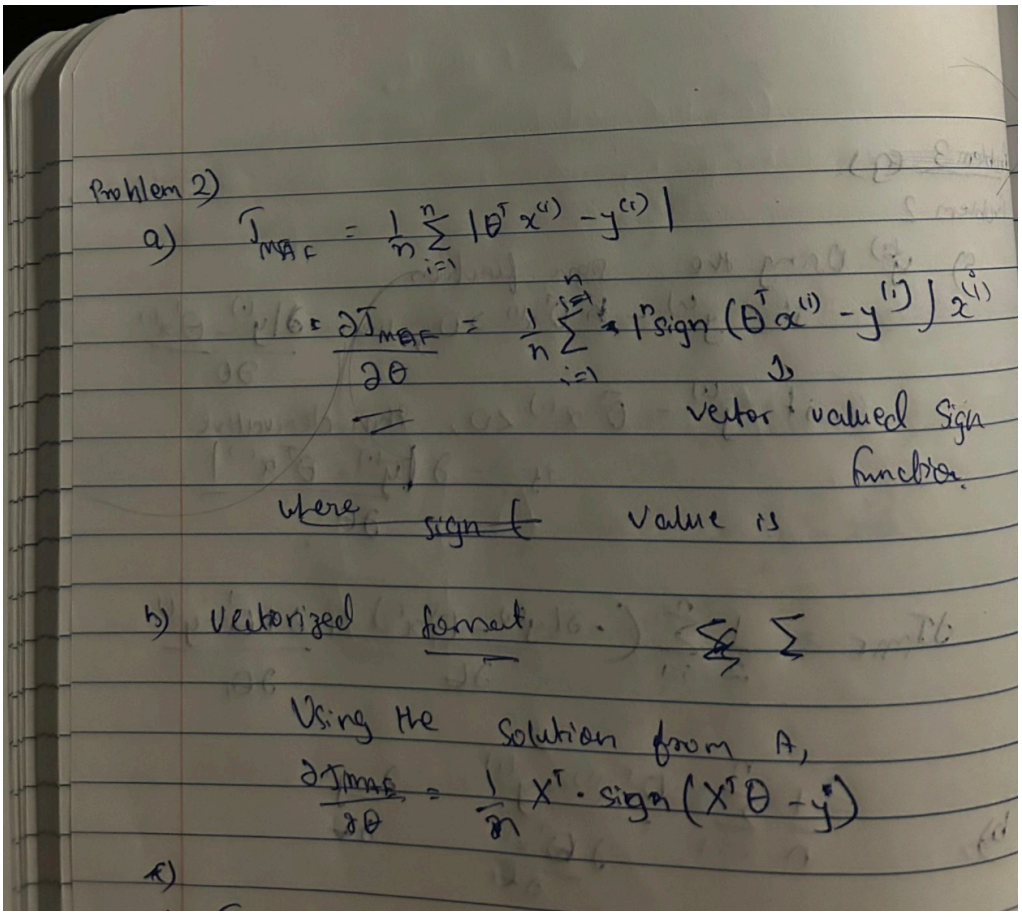
b) True: We have m values for a hyperparameter and while performing k-fold cross-validation, we train the classifier m*k times. For each of the m hyperparameter values, we split the dataset into k subsets, perform k iterations, and train the classifier with a different subset(validation set) for each iteration.

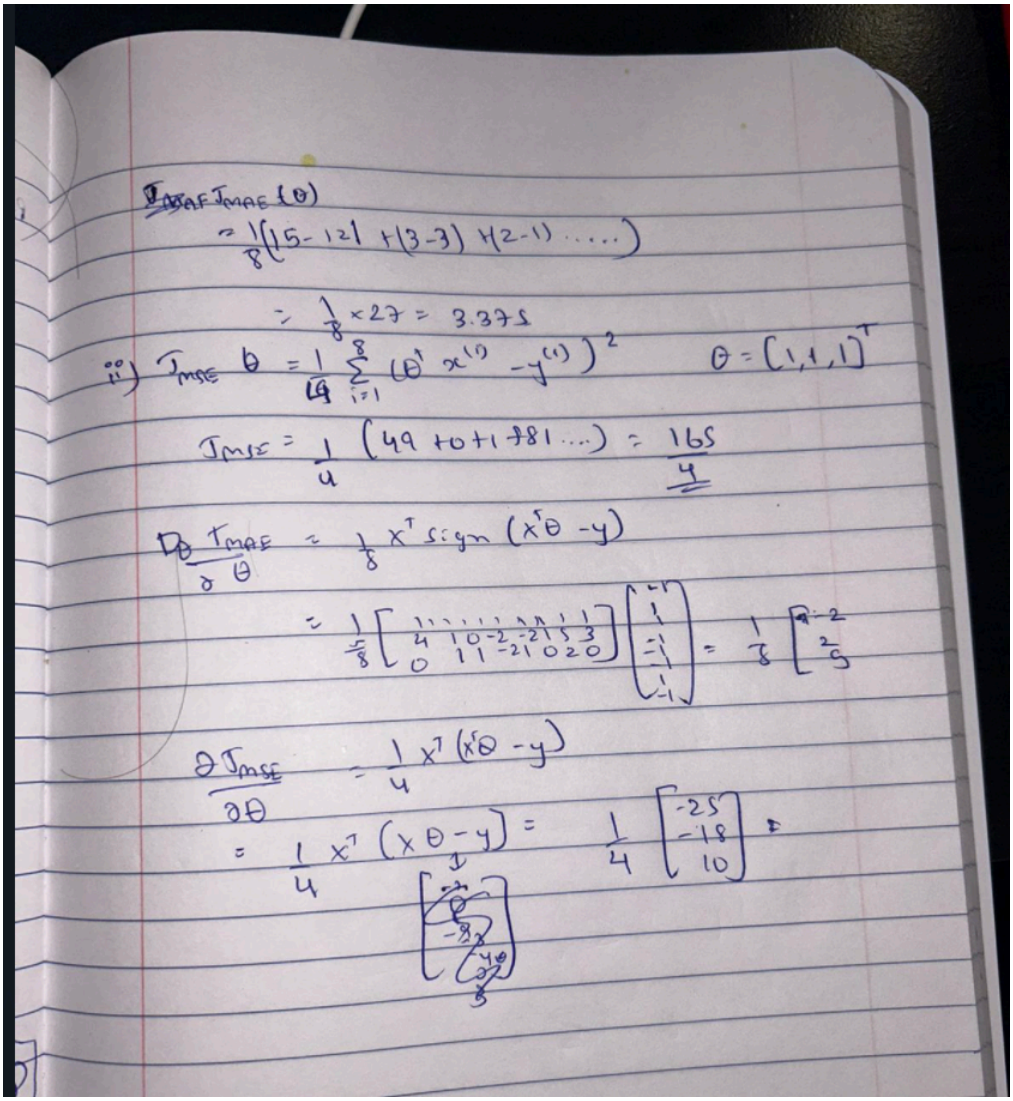
c) This would be false. The matrix $\theta^T \varphi(x)$ is not an invertible matrix, and thus the closed form solution of the matrix can't exist.

d) A: It assumes a linear relationship between the independent and dependent variable. Linear regression only works for variables that have a linear relationship, and won't work for polynomials (usually would lead to underfitting of data when using linear to model polynomial relationships). Linear models are less prone to overfitting compared to polynomial models.

e) Regularization is a great way to prevent overfitting in machine learning, as it reduces the weight of data on the loss function, thus giving a more smoother curve. The regularization term introduces a penalty on the complexity of the model by adding a term based on the magnitudes of the model parameters, discouraging them from taking extreme values.

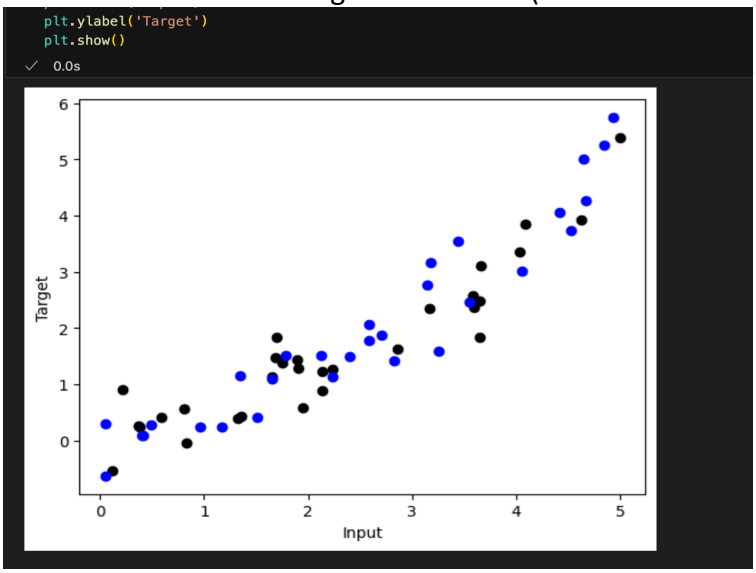
Problem 2:





Problem 3:

a) Considering the relationship between the target and the input, it seems that a linear regression model would be the best for the data. Though not exactly in a straight line (maybe exponential models would fit better for this model after a certain point where it seems to be turning exponential after input = 4), linear would be a great fit as well(linear model would have a positive gradient for theta(m*theta+bias)



b)

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

c)

```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
    if m == 1:
        # ============================================================ #
        # YOUR CODE HERE:
        # PREDICT THE TARGETS OF X
        # ============================================================ #
        X_ext = np.c_[np.ones(X.shape[0]), X]
        y_pred = np.dot(X_ext, self.theta).flatten()
    else:

        X_polynomial = self.get_poly_features(X)
        y_pred = np.dot(X_polynomial, self.theta).flatten()
        # ============================================================ #
        # END YOUR CODE HERE
        # ============================================================ #
    return y_pred
```

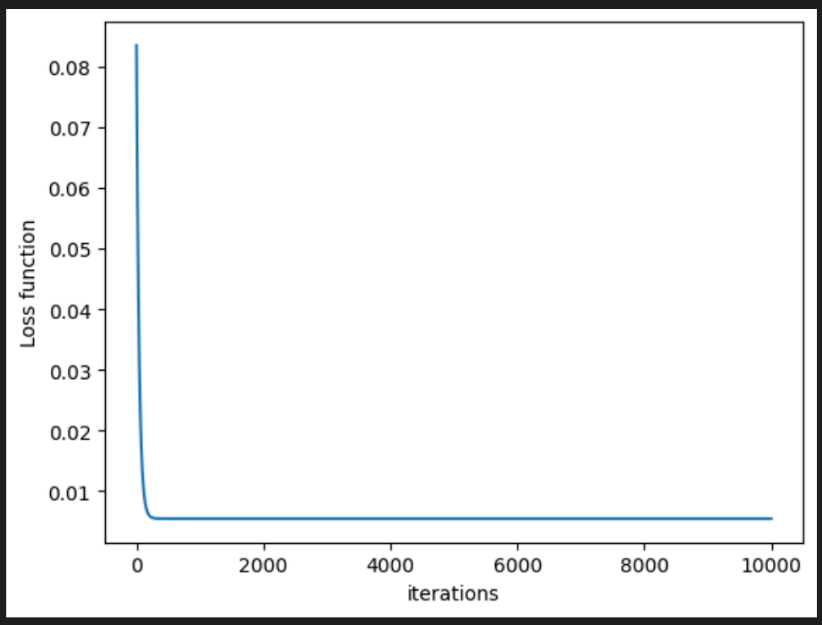
```python
## PART (c):
## Complete loss_and_grad function in Regression.py file and test your results.
regression = Regression(m=1, reg_param=0)
loss, grad = regression.loss_and_grad(X_train,y_train)
print('Loss value',loss)
print('Gradient value',grad)
##
```

✓ 0.0s

```
Loss value 1.0455416122950603
Gradient value [[1.33142275]
 [2.65167278]]
```

d)

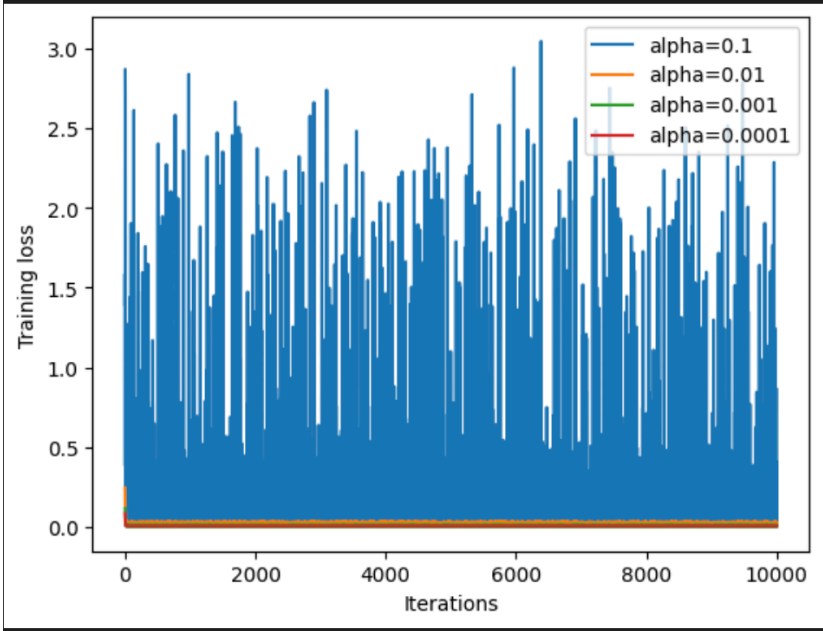
```python
## PART (d):
## Complete train_LR function in Regression.py file
loss_history, theta = regression.train_LR(X_train,y_train, alpha=1e-3, B=30, num_iters=10000)
plt.plot(loss_history)
plt.xlabel('iterations')
plt.ylabel('Loss function')
plt.show()
print(theta)
print('Final loss:',loss_history[-1])
```




```
[[0.71525596]
 [0.09894843]]
Final loss: 0.005469327327031826
```

```python
for i, B in enumerate(Bs):
    loss_history, _ = regression.train_LR(X_train, y_train, alpha=1e-2, B=B, num_iters=10000)
    losses[i, :len(loss_history)] = 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()
```

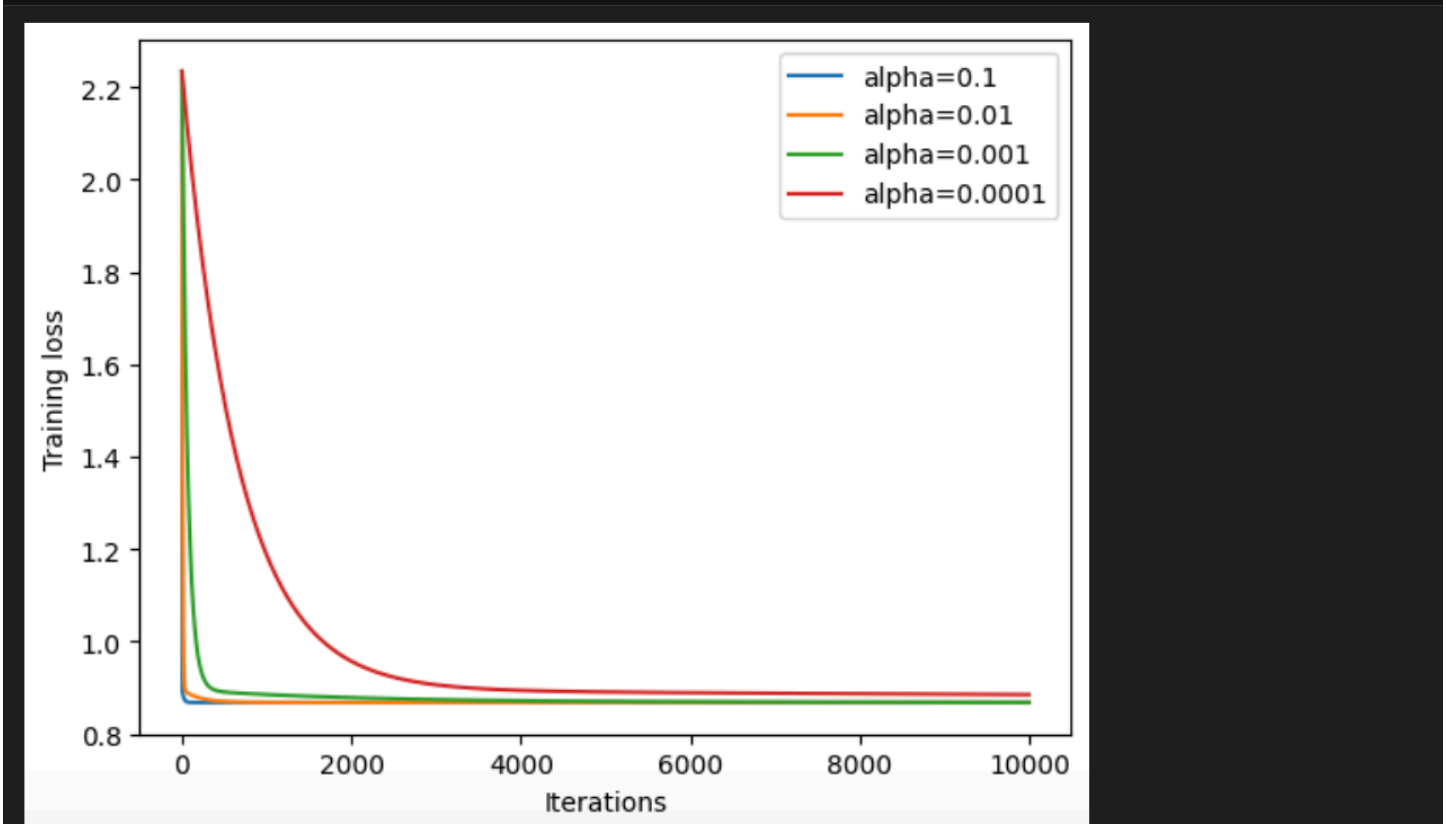
✓ 2.0s



```python
## PART (d) (Different Alpha(learning rates):
from numpy.linalg import norm
alphas = [1e-1, 1e-2, 1e-3, 1e-4]
  #losses = np.zeros((len(Bs),10000))
losses = []
for alpha in alphas:
    regression = Regression(m=1,reg_param=0)  #WE ARE USING NO REGULARIZATION!!!
    loss_history, _ = regression.train_LR(X_train, y_train, alpha=alpha, B=30, num_iters=10000)
    losses.append(loss_history)
max_length = max(len(loss_history) for loss_history in losses)
fig = plt.figure()
for i, loss in enumerate(lo
    plt.plot(range(10000), loss, label='alpha='+str(alphas[i]))
plt.xlabel('Iterations')
plt.ylabel('Training loss')
plt.legend()
plt.show()
```

(variable) loss: Any

✓ 3.5s



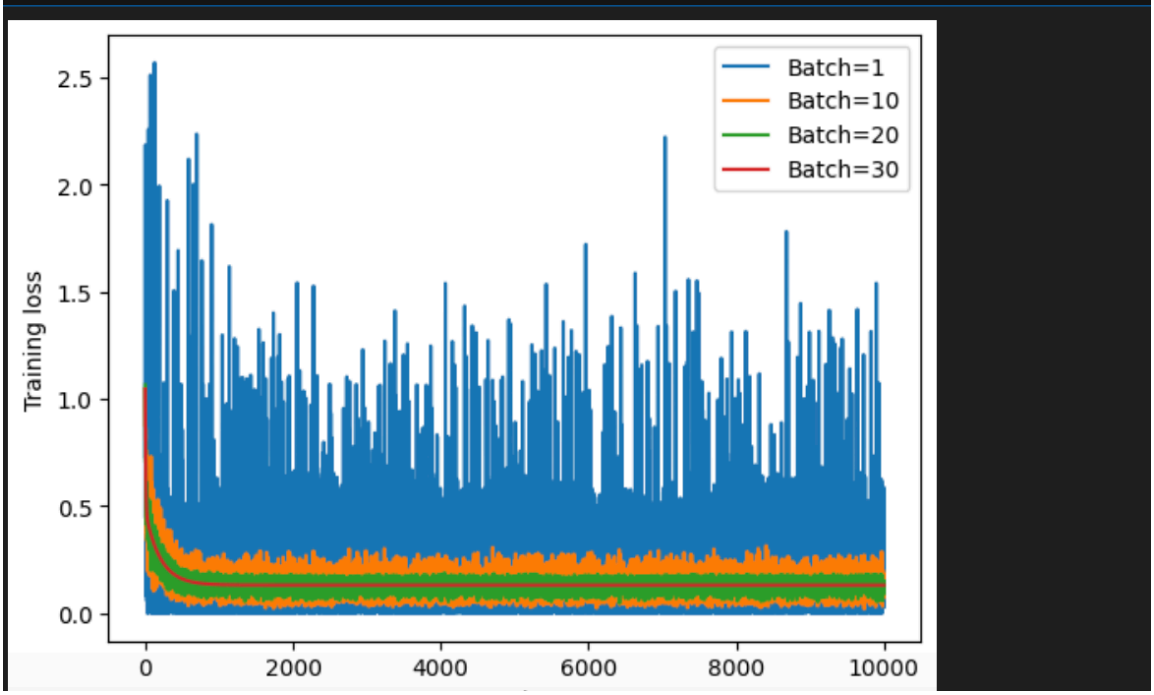
All the alphas we are using lead to convergence as shown above. This doesn't show a case where the learning rate it too high and there is no convergence. In that scenario, the alpha would too large and thus causes bouncing around rather than approaching the minima of the loss function. However, all the values we tested converge to the gradient as seen above, thus being in the optimal alpha range(though the smallest one would take more than 10000 iterations

```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
    shuff = np.column_stack((X,y))
    for t in np.arange(num_iters):
        X_batch = None
        y_batch = None

        np.random.shuffle(shuff)
        X_batch = shuff[:B, :-1]
        y_batch = shuff[:B,-1].reshape((-1,1))

        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
        # ============================================================ #
        # END YOUR CODE HERE
        # ============================================================ #
        loss_history.append(loss)
    return loss_history, self.theta
```

```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:
        y_predict = self.predict(X)
        loss = 0.5 * np.mean((y_predict - y) ** 2)
        X_ext = np.c_[np.ones(X.shape[0]), X]
        for j in range(self.theta.shape[0]):
            grad[j] = np.mean((y_predict - y) * X_ext[:, j])
        # ============================================================================ #
        # END YOUR CODE HERE
        # ============================================================================ #
    else:
        y_predict = self.predict(X)
        X_polynomial = self.get_poly_features(X)
        for i in range(self.theta.shape[0]):
            grad[i] = np.mean((y_predict - y) * X_polynomial[:, i])
            if(i != 0):
                grad[i] += 2*self.reg*self.theta[i]
        loss = 0.5 * np.mean((y_predict -      * 2)
    return loss,grad
```

```
## PART (d) (Different Learning Rates):
from numpy.linalg import norm
Bs = [1,10,20,30]
# ================================================================ #
# YOUR CODE HERE:
# Train the Linear regression for different learning rates
# ================================================================ #
for i, B in enumerate(Bs):
    loss_history, _ = regression.train_LR(X_train, y_train, alpha=1e-2, B=B, num_iters=10000)
    losses[i, :len(loss_history)] = loss_history
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
fig = plt.figure()
for i, loss in enumerate(losses):
    plt.plot(range(10000), loss, label='Batch='+str(Bs[i]))
plt.xlabel('Iterations')
plt.ylabel('Training loss')
plt.legend()
plt.show()
```

✓ 2.5s



In the above graph, there are a range of batch sizes given above(where B represents the batch size). In the case for MGDB-> there is a high loss as seen in the graph above, due to which it doesn't reach the optimal solution. The closed form solution is deterministic in nature, and thus directly computes the optimal loss for the linear regression model above

e)

```
## 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)
```

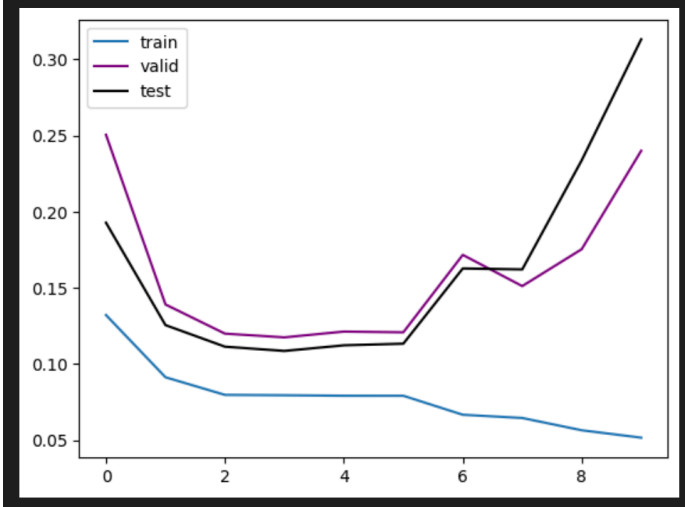
```
Optimal solution loss 0.26417938203964436
Optimal solution theta [ 0.8852483  -0.37906992]
```

```
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
    #     # ================================================================ #
    #     #print(X.shape)
    #     A = np.hstack((X, np.ones((X.shape[0],1))))
    #     #print(A.shape)
    #     self.theta = np.linalg.inv(A.T @ A) @ A.T @ y
    #     y_pred = A @ self.theta
    #     loss = np.mean((y_pred-y)**2)
    #     # ================================================================ #
    #     # END YOUR CODE HERE
    #     # ================================================================ #
    # else:
        # ================================================================ #
        # YOUR CODE HERE:
        # Extend X with get_poly_features().
        # Predict the targets of X.
        # ================================================================ #
    X_in = self.get_poly_features(X)
    self.theta = np.matmul(np.linalg.inv(np.matmul(np.transpose(X_in),X_in)),np.matmul(np.transpose(X_in), y))
    loss, grad = self.loss_and_grad(X,y)
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
    return loss, self.theta
    """
```

F)

```python
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(n):
            xi = X[i,0]
            X_out[i,0] = 1
            X_out[i, 1] = xi
        # ============================================================ #
        # 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(len(X)):
            for j in range(m+1):
                X_out[i][j] = np.power(X[i], j)
        # ============================================================ #
        # END YOUR CODE HERE
        # ============================================================ #
        #pass
    return X_out
```

```python
## PART (f):
train_loss=np.zeros((10,1))
valid_loss=np.zeros((10,1))
test_loss=np.zeros((10,1))
# ============================================================ #
for m in range(0,10):
    regression = Regression(m=m+1, reg_param=0)
    train_loss[m] = regression.closed_form(X_train, y_train)[0]
    valid_loss[m] = regression.loss_and_grad(X_valid,y_valid)[0]
    test_loss[m] = regression.loss_and_grad(X_test,y_test)[0]
# ============================================================ #
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()
```



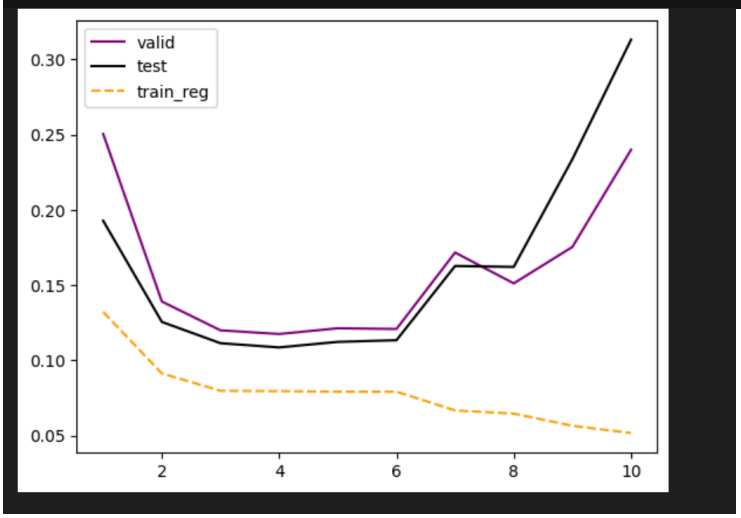
As we can see in the diagram above, higher polynomials help fit the training data better and better, thus reducing the loss, however it doesn't necessarily decrease the loss in the valid and test sets. The optimal polynomial that would model this would be of degree 2(or could be three) as it is the mimima of the test sets.

G)

```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:
        y_predict = self.predict(X)
        loss = 0.5 * np.mean((y_predict - y) ** 2)
        X_ext = np.c_[np.ones(X.shape[0]), X]
        for j in range(self.theta.shape[0]):
            grad[j] = np.mean((y_predict - y) * X_ext[:, j])
        # ============================================================ #
        # END YOUR CODE HERE
        # ============================================================ #
    else:
        y_predict = self.predict(X)
        X_polynomial = self.get_poly_features(X)
        for i in range(self.theta.shape[0]):
            grad[i] = np.mean((y_predict - y) * X_polynomial[:, i])
            if(i != 0):
                grad[i] += 2*self.reg*self.theta[i]
        loss = 0.5 * np.mean((y_predict -      * 2)
    return loss,grad
```

```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))
lambdas = [10**(-i) for i in range(10)]
for m in range(0,10):
    regression = Regression(m=m+1, reg_param=lambdas[m])
    train_reg_loss[m] = regression.closed_form(X_train, y_train)[0]
    valid_loss[m] = regression.loss_and_grad(X_valid,y_valid)[0]
    test_loss[m] = regression.loss_and_grad(X_test,y_test)[0]
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()
```

```
    __stop: SupportsIndex,
    /
)

range
```



According to this, even with higher polynomials, we don't see an increase in loss. This is due to the fact that the regression model penalizes the loss term, and thus doesn't allow the polynomial models to overfit the data. The higher the lambda, the more it penalizes the function and thus we see a decrease in overfitting.