CS M146 Fall 2023 Homework 1 Solutions

UCLA ID: 605900342
Name: Caleb Traxler
Collaborators: N/A

By turning in this assignment, I agree by the UCLA honor code and declare that all of this is my own work.

Problem 1 (Short Answer Questions)

For each of the statements below, state True (T) or False (F), or pick the correct option. Explain your answer in 1-2 sentences. For every part below, 1 point for correct assertion and 1 point for correct explanation.

(a) T/F: Ridge regression does not have a closed-form solution.

False - Ridge Regression does have a closed-form solution, hence false. Why? The ridge estimator solves the following slightly modified minimization problem

Suppose that there exits some $\lambda > 0$ constant. Also assume there exists some $k \in N$ such that

$J(\theta) = \frac{1}{2n}\sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{k=1}^{p}\theta_k^2$

$= \frac{1}{2n}\sum_{i=1}^{n}(\theta^T(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{k=1}^{p}\theta_k^2$

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

$\propto \theta^T X^T X\theta - y^T X\theta - \theta^T X^T y + y^T y + \lambda\theta^T\theta$

$\propto \theta^T X^T X\theta - 2\theta^T X^T y + y^T y + \lambda\theta^T\theta$

Upon differentiating, $\frac{\partial J(\theta)}{\partial\theta}(\theta^T X^T X\theta - 2\theta^T X^T y + y^T y + \lambda\theta^T\theta) = 0 \implies (X^T X + \lambda I^*)\theta - X^T y = 0,$

such that $I^* = \begin{pmatrix} 0 & 0 & 0 & ... & 0 \\ 0 & 1 & 0 & ... & 0 \\ 0 & 0 & 1 & ... & 0 \\ ... & ... & ... & ... & ... \\ 0 & 0 & 0 & ... & 1 \end{pmatrix}$

$\implies (X^T X + \lambda I^*)\theta = X^T y \implies \theta = (X^T X + \lambda I^*)^{-1} X^T y$

Thus the closed form solution for ridge regression exists, such that it is $\theta = (X^T X + \lambda I^*)^{-1} X^T y$.

(b) T/F: If we have m values for a hyperparameter of a classifier and we use k fold cross-validation for hyperparameter tuning, then we train the classifier mk times from scratch.

True - If we have m values for a hyperparameter of a classifier and we use k fold cross validation for hyperparameter tuning, then we indeed train the classifier mk times 'from scratch'. If you have m different values for a single hyperparameter that you want to tune, and you use k-fold cross-validation for each of those m values, then you need to train the classifier $m * k$ times. Note that this is for each candidate value of a single hyperparameter (or the set of hyperparameters if tuning more than one at the same time).

(c) T/F: Consider a linear function basis $\phi(x) = [1, logx_1, x_2^3 x_3]$, where $x \in R^3$. There exits a closed form solution for minimizing the mean squared error for the hypothesis $h_\theta(x) = \theta^T\phi(x)$. If true, state the solution. Otherwise, give a justification.

True - For the linear function basis, $\phi(x) = [1, logx_1, x_2^3 x_3]$, where $x \in R^3$. There exits a closed form solution for minimizing the mean squared error for the hypothesis $h_\theta(x) = \theta^T\phi(x)$. I know that an expression is in closed form it its formed with constants, variables, and a finite set of basic functions connected by arithmetic operations. Commonly the functions; nth roots, exponential functions and logarithms are allowed.

From the lecture notes, I know that using the linear regression cost function I can write

Suppose there exists a dataset, $(x^{(i)}, y^{(i)})\}_{i=1}^n$ which can be transformed by ϕ to each $x^{(i)}(\phi(x^{(i)}), y^{(i)})\}_{i=1}^n$

$J(\theta) = \frac{1}{2n}\sum_{i=1}^{n}(h_\theta(\phi(x^{(i)})) - (y^{(i)})^2 = \frac{1}{2n}\sum_{i=1}^{n}(h_\theta(\phi(x^{(i)})) - y^{(i)})^2$
$= \frac{1}{2n}\sum_{i=1}^{n}(\theta^T\phi(x^{(i)}) - y^{(i)})^2 = \frac{1}{2n}(X\theta - y)^T(X\theta - y)$
$\propto \theta^T X^T X\theta - y^T X\theta - \theta^T X^T y + y^T y \ \propto \theta^T X^T X\theta - 2\theta^T X^T y + y^T y$

Upon differentiating and setting the result equal to zero, I see that
$\frac{\partial}{\partial\theta}(\theta^T X^T X\theta - 2\theta^T X^T y + y^T y) = 0 \implies (X^T X)\theta - X^T y = 0 \implies \theta = (X^T X)^{-1}X^T y$
$\theta = (X^T X)^{-1}X^T y$ is a closed form solution assuming that $X^T X$ is invertible.

(d) Which of the following is a major drawback of linear regression?

A. It assumes a linear relationship between the independent and dependent variables. Since linear regression assumes a linear relationship between the independent and dependent variables, it fails to fit complex datasets properly. In most real life scenerios the relationship between the variables of the dataset isn't linear thus, a straight line cannot properly fit the data.

(e) Which of the following is a common method for preventing overfitting in machine learning?

C. Regularization. This is true by the definition of regularization, such that this technique discourages learning a more complex or flexible model, so as to avoid the risk of over-fitting. An example is ridge regression, which was talked about in (a).

Problem 2 (Mean Absolute Error)

In class, we considered optimizing the Mean Square Error (MSE) loss. In practice, there are other choices of loss functions as well. For this problem, we will consider linear regression using Mean Absolute Error (MAE) as our loss function. Specifically, the MAE loss is given as:
$J_{MAE}(\theta) = \frac{1}{n}\sum_{i=1}^{n}|\theta^T x^{(i)} - y^{(i)}|$

(a) Derive the partial derivative $\frac{\partial J_{MAE}(\theta)}{\partial\theta_j}$

Hint: For this question, you don't need to worry about $J = 0$. You can use the vector-valued sign function,
s.t $sign(\hat{y} - y) = \begin{cases} +1 & \hat{y}^{(i)} \geq y^{(i)} \\ -1 & otherwise \end{cases}$

To derive the partial derivative $\frac{\partial J_{MAE}(\theta)}{\partial\theta_j}$ I need to differentiate both sides
$\frac{\partial J_{MAE}(\theta)}{\partial\theta_j} = \frac{\partial}{\partial\theta_j}\frac{1}{n}\sum_{i=1}^{n}|\theta^T x^{(i)} - y^{(i)}|$.
I can re-write this using the vector valued sign function aswell as the following equality, $h_\theta(x^{(i)}) = \theta^T x^{(i)}$
$\implies \frac{\partial J_{MAE}(\theta)}{\partial\theta_j} = \frac{\partial}{\partial\theta_j}\frac{1}{n}\sum_{i=1}^{n} sign(h_\theta(x^{(i)}) - y^{(i)}) = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial}{\partial\theta_j}sign(h_\theta(x^{(i)}) - y^{(i)})$
Thus, using the chain rule I obtain the following
$\implies \frac{\partial J_{MAE}(\theta)}{\partial\theta_j} = \frac{1}{n}\sum_{i=1}^{n} sign(h_\theta(x^{(i)}) - y^{(i)}) * x_j^{(i)}$ or I can write
$\implies \frac{\partial J_{MAE}(\theta)}{\partial\theta_j} = \frac{1}{n}\sum_{i=1}^{n} sign(\theta^T x^{(i)} - y^{(i)}) * x_j^{(i)}$

(b) Write the vectorized solution for the gradient of the loss function, i.e. $\nabla_\theta J_{MAE}(\theta)$

Previously I found that $\frac{\partial J_{MAE}(\theta)}{\partial\theta_j} = \frac{1}{n}\sum_{i=1}^{n} sign(\theta^T x^{(i)} - y^{(i)}) * x_j^{(i)}$ is true.

I can further write the vectored solution for the gradient of the loss function as the following:
$= \nabla_\theta J_{MAE}(\theta) = \frac{1}{n}X^T sign(X\theta - y)$

(c) Given the following dataset of 8 points (listed without bias feature $x_o^{(i)}$), what is the value of $J_{MAE}(\theta)$ and $J_{MSE}(\theta)$ at $\theta = [1.0, 1.0, 1.0]^T$? How about their gradients $\nabla_\theta J_{MAE}(\theta)$ and $\nabla_\theta J_{MSE}(\theta)$?

i	1	2	3	4	5	6	7	8
$\boldsymbol{x}^{(i)}$	[4,0]	[1,1]	[0,1]	[-2,-2]	[-2,1]	[1,0]	[5,2]	[3,0]
$y^{(i)}$	12	3	1	6	3	6	8	7

Recall the following formulas

$J_{MAE}(\theta) = \frac{1}{n}\sum_{i=1}^{n}|\theta^T x^{(i)} - y^{(i)}|$, $J_{MSE}\frac{1}{2n}\sum_{i=1}^{n}(\theta^T x^{(i)} - y^{(i)})^2$

$\nabla_\theta J_{MAE}(\theta) = \frac{1}{n}X^T sign(X\theta - y)$

I will now derive $\nabla J_{MSE}(\theta)$.

$J_{MSE}(\theta) = \frac{1}{2n}\sum_{i=1}^{n}(\theta^T x^{(i)} - y^{(i)})^2$

$\implies \frac{\partial J_{MSE}}{\partial \theta_j} = \frac{\partial}{\partial \theta_j}(\frac{1}{2n}\sum_{i=1}^{n}(\theta^T x^{(i)} - y^{(i)})^2) = \frac{2}{2n}\sum_{i=1}^{n}(\theta^T x^{(i)} - y^{(i)})x_j^{(i)}$

$\implies \nabla_\theta J_{MSE} = \frac{1}{n}X^T(X\theta - y)$

Using these formulas I can now compute the values for each of the four formulas wrt the given data.

First note that I need to append a 1 to each $x^{(i)}$ to account for the bias term.

$J_{MAE}(\theta) = \frac{1}{8}(|[1,1,1]^T[1,4,0]-12|)+(|[1,1,1]^T[1,1,1]-3|)+(|[1,1,1]^T[1,0,1]-1|)+(|[1,1,1]^T[1,-2,-2]-6|) + (|[1,1,1]^T[1,-2,1]-3|) + (|[1,1,1]^T[1,1,0]-6|) + (|[1,1,1]^T[1,5,2]-8|) + (|[1,1,1]^T[1,3,0]-7|) =$

$\frac{1}{8}[|5-12| + |3-3| + |2-1| + |-3-6| + |0-3| + |2-6| + |8-8| + |4-7|] =$

$\frac{1}{8}(|-7| + |0| + |1| + |-9| + |-3| + |-4| + |0| + |-3| = \frac{27}{8} = 3.375$

So, $J_{MAE}(\theta)$ at $\theta = [1.0, 1.0, 1.0]^T$ is 3.375.

$J_{MSE}(\theta) = \frac{1}{16}([1,1,1]^T[1,4,0]-12)^2+([1,1,1]^T[1,1,1]-3)^2+([1,1,1]^T[1,0,1]-1)^2+([1,1,1]^T[1,-2,-2]-6)^2 + ([1,1,1]^T[1,-2,1]-3)^2 + ([1,1,1]^T[1,1,0]-6)^2 + ([1,1,1]^T[1,5,2]-8)^2 + ([1,1,1]^T[1,3,0]-7)^2 =$

$= \frac{1}{16}[(5-12)^2 + (3-3)^2 + (2-1)^2 + (-3-6)^2 + (0-3)^2 + (2-6)^2 + (8-8)^2 + (4-7)^2] =$

$\frac{1}{16}[(-7)^2 + (0)^2 + (1)^2 + (-9)^2 + (-3)^2 + (-4)^2 + (0)^2 + (-3)^2] = \frac{1}{16}(165) = 10.3125.$

So, $\nabla_\theta J_{MSE}(\theta)$ at $\theta = [1.0, 1.0, 1.0]^T$ is 10.3125.

$\nabla_\theta J_{MAE}(\theta) = \frac{1}{8}X^T sign(X\theta - y)$

$\implies \frac{1}{8}([1,4,0]sign(-7) + [1,1,1]sign(0) + [1,0,1]sign(1) + [1,-2,-2]sign(-9) + [1,-2,1]sign(-3) + [1,1,0]sign(-4) + [1,5,2]sign(0) + [1,3,0]sign(-3)])$

$= \frac{1}{8}([-1,-4,0] + [0,0,0] + [1,0,1] + [-1,2,2] + [-1,2,-1] + [-1,-1,0] + [0,0,0] + [-1,-3,0])$

$= \frac{1}{8}([-4,-4,2] = [-\frac{1}{2}, -\frac{1}{2}, \frac{1}{4}]$

Thus, the gradient $\nabla_\theta J_{MAE}(\theta) = [-\frac{1}{2}, -\frac{1}{2}, \frac{1}{4}]$.

$\nabla_\theta J_{MSE} = \frac{1}{n}X^T(X\theta - y)$

$\implies = \frac{1}{8}([1,4,0](-7)+[1,1,1](0)+[1,0,1](1)+[1,-2,-2](-9)+[1,-2,1](-3)+[1,1,0](-4)+[1,5,2](0)+[1,3,0](-3)])$

$= \frac{1}{8}([-7,-28,0] + [0,0,0] + [1,0,1] + [-9,18,18] + [-3,6,-3] + [-4,-4,0] + [0,0,0] + [-3,-9,0])$

$= \frac{1}{8}[-25,-17,16] = [\frac{-25}{8}, \frac{-17}{8}, \frac{16}{8}] = [-3.125, -2.125, 2]$

Thus, the gradient $\nabla_\theta J_{MSE}(\theta) = [-3.125, -2.125, 2]$.

Problem 3 (Programming Exercise: Polynomial Regression)

In this exercise, you will work through linear and polynomial regression. Our data consists of (scalar) inputs $x^{(i)} \in R$ and outputs, $y^{(i)} \in R$, $i \in \{1, ..., n\}$, which are related through a target function $y^{(i)} = f(x^{(i)})$. Your goal is to learn a linear predictor $h_\theta(x)$ that best approximates $f(x)$.

Visualization: As we learned last week, it is often useful to understand the data through visualizations. For this data set, you can use a scatter plot to visualize the data since it has only two properties to plot (x and y).

(a) Visualize the training and test data using the plot_data(...) function. Simply by looking at the dataset, can you make an educated guess on the effectiveness of linear regression in predicting the label y using the default feature x?

Based on the visualization of the training and test data, it is clear that the relationship between the feature x and the label y is not strictly linear but shows some curvature. This suggests that a simple linear regression model might not be the most effective in capturing the underlying relationship and predicting the label y accurately. However I will not completely out-rule the possibility of this data being best-predicted using a linear model.

Code (Set-up):

```
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
%load_ext autoreload
%autoreload 2
```

```
def get_data():
    """
    Load the dataset from disk and perform preprocessing to p
    """
    X_train, y_train = load('regression_train.csv')
    X_test, y_test = load('regression_test.csv')
    X_valid, y_valid = load('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)
```

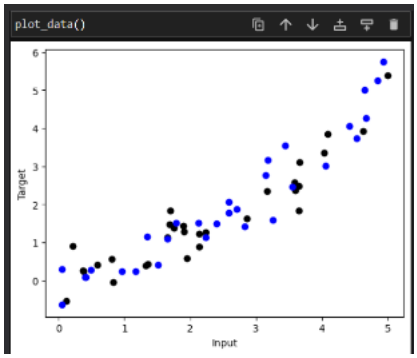
Output:

```
Train data shape:  (30, 1)
Train target shape:  (30,)
Test data shape:  (30, 1)
Test target shape:  (30,)
Valid data shape:  (20, 1)
Valid target shape:  (20,)
```

Code (Part(a)):

```python
## PART (a):
## Plot the training and test data ##
def plot_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()
```

Output:



Recall that linear regression attempts to minimize the objective function

$J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i)})^2$.

In this problem, we will use the matrix vector form where

$$\mathbf{y} = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{pmatrix}, \qquad \boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}^{(1)T} \\ \boldsymbol{x}^{(2)T} \\ \vdots \\ \boldsymbol{x}^{(n)T} \end{pmatrix}, \qquad \boldsymbol{\theta} = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{pmatrix}$$

and each instance $x^{(i)} = (1, x_1^{(i)}, ..., x_d^{(i)})^T$.

Rather than working with this fully generalized, multivariate case, let us start by considering a simple linear regression model with d = 1. Similar to what we did in class, we first augment an extra dimension $x_0 = 1$ to vectorize our hypothesis

$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$

The file regression.py contains the skeleton code for the class Regression. Objects of this class can be instantiated as model = Regression(m), where m is the degree of the polynomial feature vectors are a class of vectors having the form $[1, x_1^{(i)}, {x_1^{(i)}}^2 ..., {x_1^{(i)}}^m]^T$. Setting m = 1 instantiates a feature vector for instance i as $[1, x_1^{(i)}]^T$.

(b) Modify get_poly_features() in Regression.py for the case m=1 to create the matrix X for a simple linear model. Include a screenshot of your code in the writeup.

Code:

```python
import numpy as np
import random

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

class Regression(object):
    #default m=1 and default regularization=0 -- initialization function
    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)
    #This function gets passed X, such that X is
```

```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))


    X_out[:,0] = 1
    for i in range(1, m+1):
        X_out[:,i] = X[:,0]**i # Such that **i is the ith power

    return X_out
```

(c) Before tackling the harder problem of training the regression model, complete predict() in Regression.py() to predict y from X and θ. Include a screenshot of your code in the writeup.

Code:

```
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:
        # I know that the inputs are X... the theta vector size (m+1) contains some random real numbers
        # YOUR CODE HERE:
        y_pred = theta[0,0]+theta[1,0]*X[:,0]
    else:
        #Now I need to generalize y_pred for a general case.
        X_extend = self.get_poly_features(X) #Extending X with the function, get_poly_features()
        y_pred = np.dot(X_extend, theta)

    return y_pred
```

(d) In the lecture, we studied optimizing the loss for linear regression through gradient descent (GD) and its extension called mini-batch gradient descent (MBGD).

Recall that the parameters of our model are the θ_j values. These are the values we will iterativly adjust to minimize $J(\theta)$. In MBGD, each iteration we sample a batch of B points D_B at random from the full dataset D without replacement and perform the update

$\theta_j \leftarrow \theta_j - \alpha \frac{1}{B} \sum_{(x^{(i)}, y^{(i)}) \in D_B} (h_\theta(x^i) - y^{(i)}) x_j^{(i)}$ (simultaneously update θ_j for all j).

With each step of MBGD, we expect our updated parameters θ_j to gradually come closer to the parameters that will achieve the lowest value of $J(\theta)$.

- As we perform gradient descent, it is helpful to monitor the convergence by computing the loss, i.e. the value of the objective function J. Complete loss_and_grad() to calculate $J(\theta)$, and the gradient. Complete train_LR() and plot the loss history.

  We will use the following specifications for the gradient descent algorithm:

  - We run the algorithm for 10,000 iterations.

  - We will use a fixed step size.

  Code:

```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
    """
    n, d = X.shape
    y_pred = self.predict(X)

    loss = (1/(2*n)) * np.sum((y_pred - y)**2)

    if self.reg !=0:
        loss += (self.reg / 2) * np.sum(self.theta[1:] **2)

    regularization_term = np.zeros_like(self.theta)
    if self.reg != 0:
        regularization_term[1:] = self.reg * self.theta[1:]

    if self.m == 1:
        grad = (1/n) * X.T @ (y_pred - y) + regularization_term
    else:
        X_extend = self.get_poly_features(X)
        grad = (1/n) * X_extend.T @ (y_pred - y)+ regularization_term

    return loss, grad
```

```python
def train_LR(self, X, y, alpha=0.001, B=10, num_iters=10000) :

    ### 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
    print(self.reg)
    for t in np.arange(num_iters):
        X_batch = None
        y_batch = None

        # 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,).

        ind = np.arange(n)
        np.random.shuffle(ind)

        X_batch = X[ind[:B]]
        y_batch = y[ind[:B]]

        loss = 0.0
        grad = np.zeros_like(self.theta)

        # 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 = self.theta - alpha * grad

        #print(self.theta.shape)

        loss_history.append(loss)

    return loss_history, self.theta
```

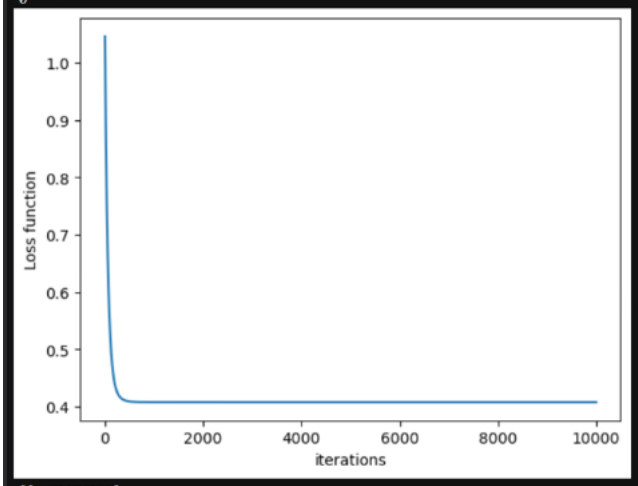
Output:

```
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)
#test = regression.predict(X_test)
#print(test)
##
```

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

```
## PART (d):
## Complete train_LR function in Regression.py file
loss_history, theta = regression.train_LR(X_train,y_train, alpha=0.001, 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
```



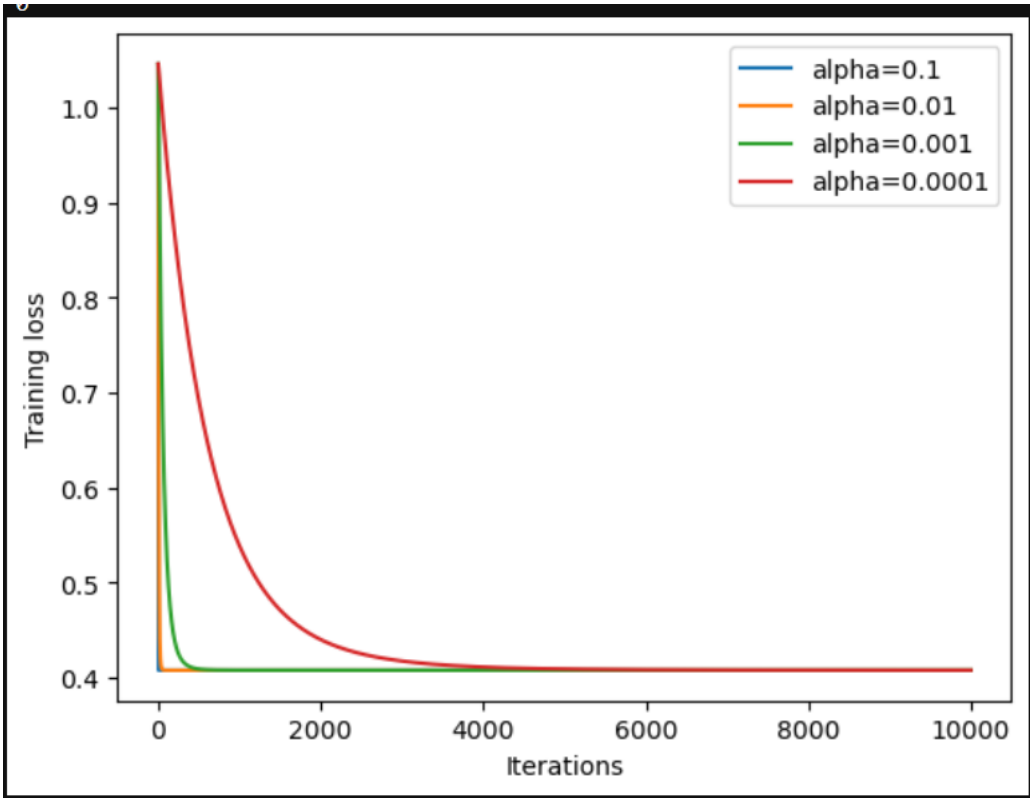

```
[[1.03974816]
 [0.42344063]]
Final loss: 0.4076510625587307
```

- So far, you have used a default learning rate(or step size) of $\alpha = 0.01$. Try different $\alpha = 10^{-4}, 10^{-3}, 10^{-1}$, and make a table of the learning rates and the final value of the objective function. Do all the learning rates lead to convergence?

  Code:

```
## 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)):
    losses_history, theta = regression.train_LR(X_train,y_train, alpha = alphas[i], B=30, num_iters=10000)
    losses[i, :] = losses_history[:10000]
# ================================================================ #
# 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()
```

  Output:

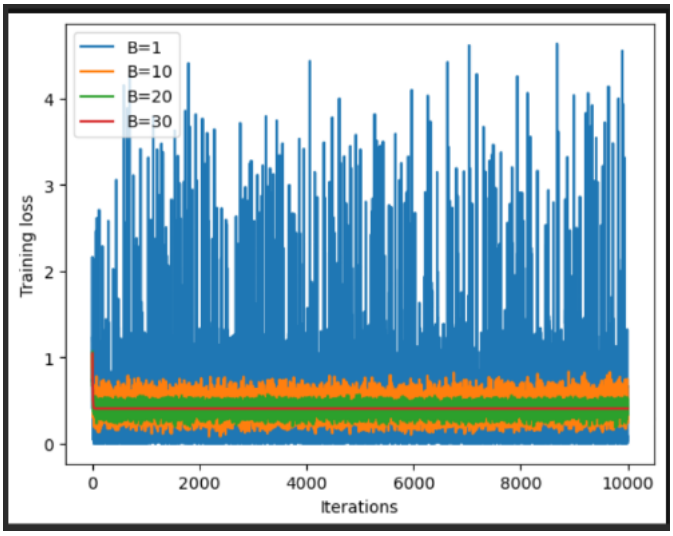


- Now lets fix $\alpha = 0.01$. Try different $B = 1, 10, 20, n$, where n is the size of the training dataset. Plot the loss history for different values of B on the same figure(i.e. 4 curves in total for B=1,10,20,n)

  Code:

```
## 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)):
    losses_history, theta = regression.train_LR(X_train,y_train, alpha=0.01, B = Bs[i], num_iters=10000)
    losses[i, :] = losses_history[:10000]
# ============================================================ #
# 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')
```

  Output:



(e) In class, we learned that the closed-form solution to linear regression is

$$\boldsymbol{\theta} = (X^T X)^{-1} X^T y.$$

Using this formula, you will get an exact solution in one step. There is no iterative repetition of parameter updates like in gradient descent.

- Implement the closed-form solution `closed_form()`.
  Code:

```
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:

        X_Transpose = np.transpose(X)

        identity = np.eye(X.shape[1])
        identity[0,0] = 0

        a = X_Transpose @ X + self.reg * identity
        invA = np.linalg.pinv(a)

        self.theta = invA @ X_Transpose @ y
        h = X @ self.theta
        loss = (1/(2*n)) * np.sum((h - y)**2)
    else:
        X_extend = self.get_poly_features(X)
        X_Transpose = np.transpose(X_extend)

        identity = np.eye(X_extend.shape[1])
        identity[0,0] = 0

        a = X_Transpose @ X_extend + self.reg * identity
        invA = np.linalg.pinv(a)

        self.theta = invA @ X_Transpose @ y
        h = X_extend @ self.theta
        loss = (1/(2*n)) * np.sum((h - y)**2)

    return loss, self.theta
```

- Run your `closed_form()` to get solution. Report the result of parameters and the loss. How do they compare to those obtained by MBGD?

  Output:

```
## 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 0.15175965470596756
Optimal solution theta [0.76186575]
```

Comparing MBGD to the closed form solution, I find that for MBGD I obtained a theta vector of ([1.03974816]; [0.42344063]] and a final loss value of 0.40765... Using the closed form solution I obtained the optimal theta, which is [0.76186575], which produced a loss of 0.1517... From this I see that the closed form solution granted more accurate theta, which resulted in a better/smaller loss. Hence, the closed form solution results in a more precise and computationally inexpensive solution to the problem.

Polynomial Regression

Now let us consider the more complicated case of polynomial regression. Here, we first define a polynomial feature map: $\phi(x) = [1, x, x^2, \cdots, x^m]^T$

Consequently, our hypothesis for linear regression is:

$$h_{\boldsymbol{\theta}}(x) = \boldsymbol{\theta}^T \phi(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_m x^m.$$

Note, here we have NOT augmented our input x with an additional dimension, as done previously, as the vectorization can be induced directly after applying the feature map ϕ.

(f) Recall that linear basis function regression (in this case, using a polynomial basis) can be considered as an extension of linear regression in which we replace our input matrix X with

$$\Phi(X) = \begin{pmatrix} \phi(x^{(1)})^T \\ \phi(x^{(2)})^T \\ \vdots \\ \phi(x^{(n)})^T \end{pmatrix},$$

where $\phi(x)$ is a vector function such that $\phi_j(x) = x^j$ for $j = 0, \ldots, m$.

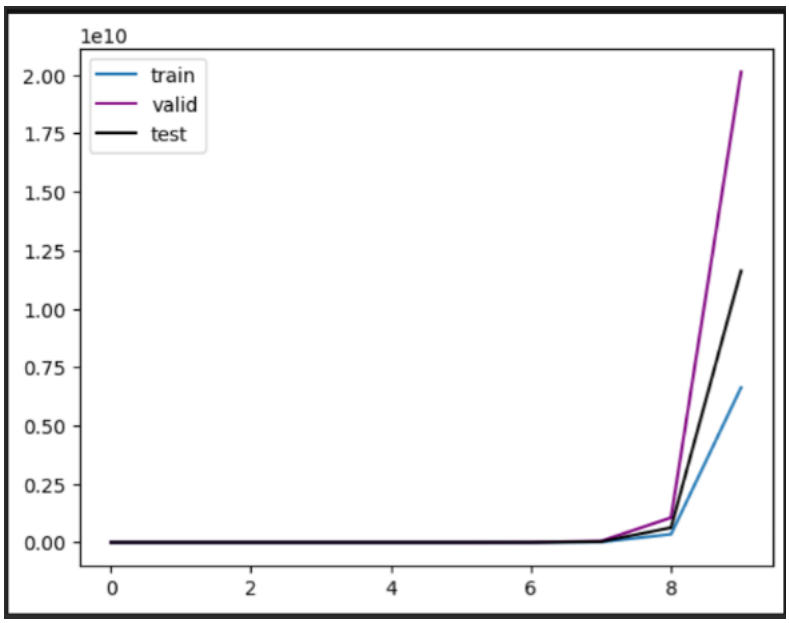
Update `get_poly_features()` for the case when $m \geq 2$. Submit a screenshot of your code for this question.

For $m = \{0, \ldots, 10\}$, use the closed-form solver to determine the best-fit polynomial regression model on the training data, and with this model, calculate the loss on both the training data and the test data. Generate a plot depicting how loss varies with model complexity (polynomial degree) – you should generate a single plot with both training, validation and test error, and include this plot in your writeup. Which degree polynomial would you say best fits the data? Was there evidence of under/overfitting the data? Use your plot to justify your answer.

Code:

```python
## 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(10):
    regression = Regression(m=m, reg_param=0)

    x_train_poly = regression.get_poly_features(X_train)
    x_valid_poly = regression.get_poly_features(X_valid)
    x_test_poly = regression.get_poly_features(X_test)

    _, theta = regression.closed_form(x_train_poly, y_train)

    train_loss[m] = np.mean((np.dot(x_train_poly, theta ) - y_train)**2)
    valid_loss[m] = np.mean((np.dot(x_valid_poly, theta ) - y_valid)**2)
    test_loss[m] = np.mean((np.dot(x_test_poly, theta ) - y_test)**2)
# ============================================================ #
# 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()
```

Output:



Regularization

Finally, we will explore the role of regularization. For this problem, we will use L_2-regularization so that our regularized objective function is

$$J(\boldsymbol{\theta}) = \frac{1}{2n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2}||\boldsymbol{\theta}_{[1:m]}||^2,$$

again optimizing for the parameters $\boldsymbol{\theta}$.

(g) Modify `loss_and_grad()` to incorporate ℓ_2-regularization using the closed-form solution. Submit a screenshot of your code for this question. Use your updated solver to find the coefficients that minimize the error for a tenth-degree polynomial ($m = 10$) given regularization factor $\lambda = 0, 10^{-8}, 10^{-7}, \ldots, 10^{-1}, 10^{0}$. Now we want to check how λ affects the loss (unregularized) on the training data, validation data, and test data. Generate a plot depicting how the loss error varies with λ (for your x-axis, let $x = [1, 2, \ldots, 10]$ correspond to $\lambda = [0, 10^{-8}, 10^{-7}, \ldots, 10^{0}]$ so that λ is on a logarithmic scale, with regularization increasing as x increases), and include this plot in your writeup. Which λ value appears to work best?

$||\theta_{[1:m]}||^2 = \sum_{i=1}^{m}\theta_i^2$, thus I can write that the loss function is the following function.

$J(\boldsymbol{\theta}) = \frac{1}{2n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2}\sum_{i=1}^{m}\theta_i^2$

Upon differentiating the loss function I find the following

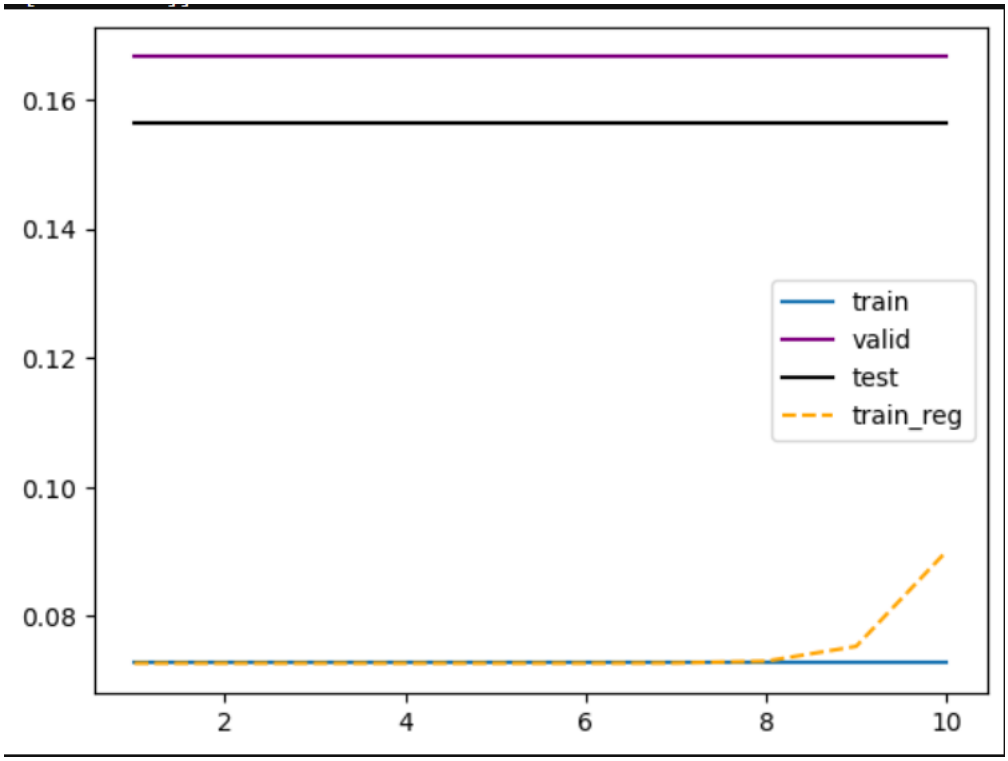
$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_j} = \frac{\partial}{\partial \theta_j}(\frac{1}{2n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2}\sum_{i=1}^{m}\theta_i^2)$

$= \frac{\partial}{\partial \theta_j}(\frac{1}{2n}\sum_{i=1}^{n} h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)}\big)^2) + \frac{\partial}{\partial \theta_j}(\frac{\lambda}{2}\sum_{i=1}^{m}\theta_i^2) = \frac{1}{n}\sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})x_j + \lambda\theta_j$

Code:

```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
# ============================================================ #
from codes.Regression import Regression

lambdas = [0, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e0]
for i in range(10):
    regression = Regression(m=10, reg_param = 0)

    regression.closed_form(X_train, y_train)

    train_loss[i], _ = regression.loss_and_grad(X_train, y_train)
    valid_loss[i], _ = regression.loss_and_grad(X_valid, y_valid)
    test_loss[i], _ = regression.loss_and_grad(X_test, y_test)


    regression = Regression(m=10, reg_param = lambdas[i])

    regression.closed_form(X_train, y_train)

    train_reg_loss[i], _ = regression.loss_and_grad(X_train, y_train)


# ============================================================ #
# 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()
```

Output:


0.16
0.14
0.12
0.10
0.08
2
4
6
8
10
train
valid
test
train_reg