Source Codes

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
```

```
            # ================================================================ #
        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
```