```python
import numpy as np
from tqdm import tqdm

class Linear_Regression():
    def __init__(self, alpha = 1e-3 , num_iter = 10000, early_stop = 100,
        intercept = True, init_weight = None, penalty = None,
        lam = 1e-3, normalize = False, adaptive=True):
        """
        Linear Regression with gradient descent method.

        Attributes:
        -------------
        alpha: Learning rate.
        num_iter: Number of iterations
        early_stop: Number of steps without improvements
                    that triggers a stop training signal
        intercept: True = with intercept (bias), False otherwise
        init_weight: Optional. The initial weights passed into the model,
                                for debugging
        penalty: {None,l1,l2}. Define regularization type for regression.
                  None: Linear Regression
                  l1: Lasso Regression
                  l2: Ridge Regression
        lam: regularization constant.
        normalize: True = normalize data, False otherwise
        adaptive: True = adaptive learning rate, False = fixed learning rate
        """
        self.alpha = alpha
        self.num_iter = num_iter
        self.early_stop = early_stop
        self.intercept = intercept
        self.init_weight = init_weight
        self.penalty = penalty
        self.lam = lam
        self.normalize = normalize
        self.adaptive = adaptive

    def fit(self, X, y):
        # initialize X, y
        self.X = X
        self.y = np.array([y]).T

        self.max = np.zeros((1, X.shape[1]))
        self.min = np.zeros((1, X.shape[1]))
        for i in range(self.X.shape[1]):
            self.max[:, i] = self.X[:, i].max()
            self.min[:, i] = self.X[:, i].min()

        ############### START TODO 1 ###############
        # Normalize the data using the formula provided in lecture
        if self.normalize:
            self.X = (self.X - self.min) / (self.max - self.min)
        ############### END TODO 1 ###############

        ############### START TODO 2 ###############
        # Add bias (if necessary) by concatanating a constant column into X
        # Hint: go through HW1 Q5 might be helpful
        if self.intercept:
            col = np.ones((len(self.X), 1))
            self.X = np.hstack((self.X, col))
        ############### END TODO 2 ###############

        # initialize coefficient
        self.coef = self.init_weight if self.init_weight is not None\
        else np.array([np.random.uniform(-1,1,self.X.shape[1])]).T

        # start training, self.loss is used to record losses over iterations
        self.loss = []
        self.gradient_descent()

    def gradient(self):
        coef = -2 / len(self.X)

        ############### START TODO 3 ###############
        # Find prediction and gradient
```

```python
        # Hint: Find the model's prediction from the given inputs with the
        #       coefficient, then calculate the gradient
        # If you forgot the formula, find them in lecture 4 and 5
        pred = self.X @ self.coef
        grad = coef * (self.X.T @ (self.y - pred))
        ############### END TODO 3 ###############

        ############### START TODO 4 ###############
        # Implement regularization penalty
        # Hint: Use self.lam
        if self.penalty == 'l2':
            grad += 2 * self.lam * self.coef
        elif self.penalty == 'l1':
            grad += self.lam * np.sign(self.coef)
        else:
            pass
        ############### END TODO 4 ###############

        return grad

    def gradient_descent(self):
        print('Start Training')
        for i in range(self.num_iter):

            ############### START TODO 5 ###############
            # calculate prediction y based on current coefficients (self.coef)
            previous_y_hat = self.X @ self.coef
            grad = self.gradient()
            # calculate the new coefficients after incorporating the gradient
            temp_coef = self.coef - (self.alpha * grad)
            ############### END TODO 5 ###############

            ############### START TODO 6 ###############
            # calculate regularization cost (alias: regularization loss) based on
            # self.coef and temp_coef
            if self.penalty == 'l2':
                previous_reg_cost = self.lam * np.sum(np.square(self.coef))
                current_reg_cost = self.lam * np.sum(np.square(temp_coef))
            elif self.penalty == 'l1':
                previous_reg_cost = self.lam * np.sum(np.abs(self.coef))
                current_reg_cost = self.lam * np.sum(np.abs(temp_coef))
            else:
                previous_reg_cost = 0
                current_reg_cost = 0
            ############### END TODO 6 ###############

            ############### START TODO 7 ###############
            # Calculate error (alias: loss) using sum squared loss
            # and add regularization cost
            pre_error = np.sum(np.square( self.y - previous_y_hat )) + previous_reg_cost
            current_y_hat = self.X @ temp_coef
            current_error = np.sum(np.square( self.y - current_y_hat )) + current_reg_cost
            ############### END TODO 7 ###############

            # Early Stop: early stop is triggered if loss is not decreasing
            # for some number of iterations
            if len(self.loss) > self.early_stop and \
            self.loss[-1] >= max(self.loss[-self.early_stop:]):
                print('-----------------------')
                print(f'End Training (Early Stopped at iteration {i})')
                return self

            ############### START TODO 8 ###############
            # Implement adaptive learning rate

            # Rules: if current error is smaller than previous error,
            # multiply the current learning rate by 1.3 and update coefficients,
            # otherwise by 0.9 and do nothing with coefficients
            if current_error < pre_error:
                self.alpha = 1.3 * self.alpha if self.adaptive else self.alpha
                self.coef = temp_coef
            else:
                self.alpha = 0.9 * self.alpha if self.adaptive else self.alpha
            ############### END TODO 8 ###############
```

```
            # record stats
            self.loss.append(float(current_error))
            if i % 1000000 == 0:
                print('-----------------------')
                print('Iteration: ' +  str(i))
                print('Coef: '+ str(self.coef))
                print('Loss: ' + str(current_error))
        print('-----------------------')
        print('End Training')
        return self

    def predict(self, X):
        X_norm = np.zeros(X.shape)
        for i in range(X.shape[1]):
            X_norm[:, i] = (X[:, i] - self.min[:, i]) / (self.max[:, i] - self.min[:, i])
        X = X_norm
        ############### START TODO 9 ###############
        # add bias (if necessary, same as TODO 2)
        if self.intercept:
            col = np.ones((len(X), 1))
            X = np.hstack((X, col))

        # Find the model's predictions
        # Hint: Use matrix multiplication ('@' might come in handy here)
        y = X @ self.coef
        return y
        ############### END TODO 9 ###############

    # Congrats! You have reached the end of this model's implementation :)
```