```python
from numpy import *

class HW3():

    def __init__(self):
        self.data = genfromtxt("crime-train.txt", names=True, dtype=float)

        # Do all of this just to get our data into matrix form...
        test = []
        for x in self.data.dtype.names:
            test.append(self.data[x])

        test.append(ones(1595)) #include column of all ones
        self.trainData = array(test)
        self.trainData = transpose(self.trainData)

        self.t_data = genfromtxt("crime-test.txt", names=True, dtype=float)

        # Do all of this just to get our data into matrix form...
        test = []
        for x in self.t_data.dtype.names:
            test.append(self.t_data[x])

        test.append(ones(399))  # Add intercept term
        self.testData = array(test)
        self.testData = transpose(self.testData)

    def linearRegression(self):
        """
        This function is the implementation of the linear regression model. It splits the training/test data set
        into matrix X of features and vector Y of outputs.

        The weight vector is calculated as (X.T * X)^-1 * X.T * Y

        Which is then utilized as the weight vector for the prediction model of the test data set.

        The RSME value is calculated using the costFunction().
        :return: None
        """
        Y = self.trainData[:,0]
        X = self.trainData[:,1:]
        Y_i = self.testData[:,0]
        X_i = self.testData[:,1:]
```

```

        W = dot(dot(linalg.inv(dot(transpose(X),X)),transpose(X)),Y)
        predict = dot(X_i,transpose(W))
        RSME = self.costFunction(Y_i, predict, 399)

        print("Linear Regression RSME: {}".format(RSME))

    def linearRegressionGradientDescent(self):
        """
        This function is the implementation of the linear regression model - gradient descent version.

        Parameters are initalized to control the model, most importantlyy:
        alpha - our step size
        eps - our loss tolerance

        theta - Our weight vector, initialized as a random vector of Gaussian distribution.
        :return: None.
        """
        Y = self.trainData[:, 0]
        X = self.trainData[:, 1:]

        Y_i = self.testData[:, 0]
        X_i = self.testData[:, 1:]

        grad = None
        theta = random.normal(0,1,96)
        alpha = 0.01
        eps = float('1e-7')

        previousLoss = self.costFunction(Y, None, 1595, theta, X)
        loss = None

        while True:
            # Use training data to calculate the gradient
            h_x = dot(X, theta)
            grad = theta + ( (alpha * dot(transpose(X), (Y - h_x))) / 1595)

            # Use training data to calculate the square error loss
            loss = self.costFunction(Y, None, 1595, grad, X)

            if abs(loss - previousLoss) < eps: # If |L(w+1) - L(w)| < 0.0001
                break # Break out of the cycle.
```

```
            theta = grad # Update our weight vector
            previousLoss = loss # Set previousLoss to currentLoss so we don't have to recalculate this.

        print("Linear Regression Gradient Descent RSME: {}".format(self.costFunction(Y_i, None, 399, grad, X_i)))

    def ridgeRegressionGradientDescent(self):
        """
        This function is the implementation of the ridge regression model - gradient descent version.

        Parameters are initalized to control the model, most importantlyy:
        alpha - our step size
        eps - our loss tolerance

        theta - Our weight vector, initialized as a random vector of Gaussian distribution.
        :return: None.
        """

        Y = self.trainData[:, 0]
        X = self.trainData[:, 1:]

        Y_i = self.testData[:, 0]
        X_i = self.testData[:, 1:]

        grad = None
        theta = random.normal(0,1,96)
        alpha = 0.01
        eps = float('1e-7')

        previousLoss = self.costFunction(Y, None, 1595, theta, X)
        loss = None

        regTerm = self.crossValidation()

        while True:
            h_x = dot(X, theta)
            grad = theta + ((alpha * (dot(transpose(X), Y - h_x) - regTerm * theta)) / 1595)

            loss = self.costFunction(Y, None, 1595, grad, X)
            if abs(loss - previousLoss) < eps: # If |L(w+1) - L(w)| < 0.0001
```

```python
                    break # Break out of the cycle.

                theta = grad  # Update our weight vector
                previousLoss = loss  # Set previousLoss to currentLoss so we don't have to recalculate this.

        print("Ridge Regression Gradient Descent RSME: {}".format(self.costFunction(Y_i, None, 399, grad, X_i)))

    def costFunction(self, Y_i, predict, n, W = None, X_i = None):
        """
        This function calculate the square loss for a given set of truth values and predicted values.
        :param Y_i: <vector> - A vector of accepted true values.
        :param predict:  <vector> - A vector of predicted values
        :param n: <int> - The number of data points used.
        :param W:  <vector> - A vector of weights used to calculate the prediction.
        :param X_i: <matrix> - A matrix of feature values used to calculate the prediction
        :return: RSME: <float> - The root-square loss
        """

        # We can pass in W and X_i if we want the costFunction to just calculate the prediction for us.
        if W is not None and X_i is not None:
            predict = dot(X_i, transpose(W))

        # Calculate the RSME.
        RSME = sqrt(sum(subtract(Y_i, predict) ** 2) / n)

        return RSME

    def ridgeRegression(self):
        """
        This function is the implementation of the ridge regression model. It splits the training/test data set
        into matrix X of features and vector Y of outputs.

        The weight vector is calculated as (X.T * X + Lambda*I )^-1 * X.T * Y

        Which is then utilized as the weight vector for the prediction model of the test data set.

        The RSME value is calculated using the costFunction().
        :return: None
```

```python
        """

        regTerm = self.crossValidation()

        Y = self.trainData[:, 0]
        X = self.trainData[:, 1:]
        Y_i = self.testData[:, 0]
        X_i = self.testData[:, 1:]

        W = dot(dot(linalg.inv(dot(transpose(X), X) + regTerm * identity(96)), transpose(X)), Y)

        predict = dot(X_i, transpose(W))
        RSME = self.costFunction(Y_i, predict, 399)

        print("Ridge Regression RSME: {}".format(RSME))

    def ridgeRegressionBase(self, X, Y, regTerm, test):
        W = dot(dot(linalg.inv(dot(transpose(X),X) + regTerm*identity(96)),transpose(X)),Y)

        X_i = test[:,1:]
        Y_i = test[:,0]

        predict = dot(X_i, transpose(W))
        RSME = self.costFunction(Y_i, predict, 319)

        return RSME

    def crossValidation(self):
        """
        This function performs a k-fold validation to determine the lambda value of a ridgeRegression model via repeatedly
        testing a particular lambda value for a given training set. The average RSME of a particular k trial cycle
        is used to determine if the lambda value for that cycle is the one that produces the smallest average RSME.

        :return: minRegTerm: <float> - The lambda value that produces the smallest average RSME value.
        """
        segOne = self.trainData[0:319,:]
        segTwo = self.trainData[319:638,:]
        segThree = self.trainData[638:957,:]
        segFour = self.trainData[957:1276,:]
        segFive = self.trainData[1276:1595,:]

```

```
        regTerm = 400
        minRegTerm = None
        minRSME = None

        for k in range(10):

            kOneData = concatenate((segOne, segTwo, segThree, segFour))

            kOneRSME = self.ridgeRegressionBase(kOneData[:,1:], kOneData[:,0], regTerm, segFive)

            kTwoData = concatenate((segOne, segTwo, segThree, segFive))

            kTwoRSME = self.ridgeRegressionBase(kTwoData[:,1:], kTwoData[:,0], regTerm, segFour)

            kThreeData = concatenate((segOne, segTwo, segFour, segFive))

            kThreeRSME = self.ridgeRegressionBase(kThreeData[:, 1:], kThreeData[:, 0], regTerm, segThree)

            kFourData = concatenate((segOne, segThree, segFour, segFive))

            kFourRSME = self.ridgeRegressionBase(kFourData[:, 1:], kFourData[:, 0], regTerm, segTwo)

            kFiveData = concatenate((segTwo, segThree, segFour, segFive))

            kFiveRSME = self.ridgeRegressionBase(kFiveData[:, 1:], kFiveData[:, 0], regTerm, segOne)


            avgRSME = (kOneRSME + kTwoRSME + kThreeRSME + kFourRSME + kFiveRSME) / 5

            if minRSME is None:
                minRSME = avgRSME
                minRegTerm = regTerm
            if avgRSME <= minRSME:
                minRSME = avgRSME
                minRegTerm = regTerm

            regTerm /= 2

        return minRegTerm

if __name__ == "__main__":
```

```
    t = HW3()

    t.linearRegression()
    t.linearRegressionGradientDescent()
    t.ridgeRegression()
    t.ridgeRegressionGradientDescent()
```