

```

1 from numpy import *
2
3 class HW3():
4
5     def __init__(self):
6         self.data = genfromtxt("crime-train.txt", names=True,
7 dtype=float)
8
9         # Do all of this just to get our data into matrix form...
10        test = []
11        for x in self.data.dtype.names:
12            test.append(self.data[x])
13
14        test.append(ones(1595)) #include column of all ones
15        self.trainData = array(test)
16        self.trainData = transpose(self.trainData)
17
18        self.t_data = genfromtxt("crime-test.txt", names=True,
19 dtype=float)
20
21        # Do all of this just to get our data into matrix form...
22        test = []
23        for x in self.t_data.dtype.names:
24            test.append(self.t_data[x])
25
26        test.append(ones(399)) # Add intercept term
27        self.testData = array(test)
28        self.testData = transpose(self.testData)
29
30    def linearRegression(self):
31        """
32        This function is the implementation of the linear regression
33        model. It splits the training/test data set
34        into matrix X of features and vector Y of outputs.
35
36        The weight vector is calculated as  $(X.T * X)^{-1} * X.T * Y$ 
37
38        Which is then utilized as the weight vector for the prediction
39        model of the test data set.
40
41        The RSME value is calculated using the costFunction().
42        :return: None
43        """
44        Y = self.trainData[:,0]
45        X = self.trainData[:,1:]
46        Y_i = self.testData[:,0]
47        X_i = self.testData[:,1:]

```

```

44
45     W = dot(dot(linalg.inv(dot(transpose(X),X)),transpose(X)),Y)
46     predict = dot(X_i,transpose(W))
47     RSME = self.costFunction(Y_i, predict, 399)
48
49     print("Linear Regression RSME: {}".format(RSME))
50
51     def linearRegressionGradientDescent(self):
52         """
53         This function is the implementation of the linear regression
model – gradient descent version.
54
55         Parameters are initalized to control the model, most
importantly:
56         alpha – our step size
57         eps – our loss tolerance
58
59         theta – Our weight vector, initialized as a random vector of
Gaussian distribution.
60         :return: None.
61         """
62         Y = self.trainData[:, 0]
63         X = self.trainData[:, 1:]
64
65         Y_i = self.testData[:, 0]
66         X_i = self.testData[:, 1:]
67
68         grad = None
69         theta = random.normal(0,1,96)
70         alpha = 0.01
71         eps = float('1e-7')
72
73         previousLoss = self.costFunction(Y, None, 1595, theta, X)
74         loss = None
75
76         while True:
77             # Use training data to calculate the gradient
78             h_x = dot(X, theta)
79             grad = theta + ( (alpha * dot(transpose(X), (Y - h_x))) /
1595)
80
81             # Use training data to calculate the square error loss
82             loss = self.costFunction(Y, None, 1595, grad, X)
83
84             if abs(loss - previousLoss) < eps: # If |L(w+1) - L(w)| <
0.0001
85                 break # Break out of the cycle.

```

```

86
87         theta = grad # Update our weight vector
88         previousLoss = loss # Set previousLoss to currentLoss so we
don't have to recalculate this.
89
90         print("Linear Regression Gradient Descent RSME:
{}".format(self.costFunction(Y_i, None, 399, grad, X_i)))
91
92     def ridgeRegressionGradientDescent(self):
93         """
94         This function is the implementation of the ridge regression model
- gradient descent version.
95
96         Parameters are initialized to control the model, most
importantly:
97         alpha - our step size
98         eps - our loss tolerance
99
100        theta - Our weight vector, initialized as a random vector of
Gaussian distribution.
101        :return: None.
102        """
103
104        Y = self.trainData[:, 0]
105        X = self.trainData[:, 1:]
106
107        Y_i = self.testData[:, 0]
108        X_i = self.testData[:, 1:]
109
110        grad = None
111        theta = random.normal(0,1,96)
112        alpha = 0.01
113        eps = float('1e-7')
114
115        previousLoss = self.costFunction(Y, None, 1595, theta, X)
116        loss = None
117
118        regTerm = self.crossValidation()
119
120        while True:
121            h_x = dot(X, theta)
122            grad = theta + ((alpha * (dot(transpose(X), Y - h_x) -
regTerm * theta)) / 1595)
123
124            loss = self.costFunction(Y, None, 1595, grad, X)
125            if abs(loss - previousLoss) < eps: # If  $|L(w+1) - L(w)| <
0.0001$ 
```

```

126         break # Break out of the cycle.
127
128         theta = grad # Update our weight vector
129         previousLoss = loss # Set previousLoss to currentLoss so we
    don't have to recalculate this.
130
131         print("Ridge Regression Gradient Descent RSME:
132 {}").format(self.costFunction(Y_i, None, 399, grad, X_i))
133
134     def costFunction(self, Y_i, predict, n, W = None, X_i = None):
135         """
136         This function calculate the square loss for a given set of truth
137         values and predicted values.
138         :param Y_i: <vector> – A vector of accepted true values.
139         :param predict: <vector> – A vector of predicted values
140         :param n: <int> – The number of data points used.
141         :param W: <vector> – A vector of weights used to calculate the
142         prediction.
143         :param X_i: <matrix> – A matrix of feature values used to
144         calculate the prediction
145         :return: RSME: <float> – The root-square loss
146         """
147
148         # We can pass in W and X_i if we want the costFunction to just
149         calculate the prediction for us.
150         if W is not None and X_i is not None:
151             predict = dot(X_i, transpose(W))
152
153         # Calculate the RSME.
154         RSME = sqrt(sum(subtract(Y_i, predict) ** 2) / n)
155
156         return RSME
157
158     def ridgeRegression(self):
159         """
160         This function is the implementation of the ridge regression
161         model. It splits the training/test data set
162         into matrix X of features and vector Y of outputs.
163
164         The weight vector is calculated as  $(X.T * X + \text{Lambda} * I)^{-1} * X.T$ 
165         * Y
166
167         Which is then utilized as the weight vector for the prediction
168         model of the test data set.
169
170         The RSME value is calculated using the costFunction().
171         :return: None

```

```

164         """
165
166         regTerm = self.crossValidation()
167
168         Y = self.trainData[:, 0]
169         X = self.trainData[:, 1:]
170         Y_i = self.testData[:, 0]
171         X_i = self.testData[:, 1:]
172
173         W = dot(dot(linalg.inv(dot(transpose(X), X) + regTerm *
identity(96))), transpose(X)), Y)
174
175         predict = dot(X_i, transpose(W))
176         RSME = self.costFunction(Y_i, predict, 399)
177
178         print("Ridge Regression RSME: {}".format(RSME))
179
180     def ridgeRegressionBase(self, X, Y, regTerm, test):
181         W = dot(dot(linalg.inv(dot(transpose(X), X) +
regTerm*identity(96))),transpose(X)),Y)
182
183         X_i = test[:,1:]
184         Y_i = test[:,0]
185
186         predict = dot(X_i, transpose(W))
187         RSME = self.costFunction(Y_i, predict, 319)
188
189         return RSME
190
191     def crossValidation(self):
192         """
193         This function performs a k-fold validation to determine the
lambda value of a ridgeRegression model via repeatedly
194         testing a particular lambda value for a given training set. The
average RSME of a particular k trial cycle
195         is used to determine if the lambda value for that cycle is the
one that produces the smallest average RSME.
196
197         :return: minRegTerm: <float> – The lambda value that produces the
smallest average RSME value.
198         """
199         segOne = self.trainData[0:319,:]
200         segTwo = self.trainData[319:638,:]
201         segThree = self.trainData[638:957,:]
202         segFour = self.trainData[957:1276,:]
203         segFive = self.trainData[1276:1595,:]
204

```

```

205         regTerm = 400
206         minRegTerm = None
207         minRSME = None
208
209         for k in range(10):
210
211             kOneData = concatenate((segOne, segTwo, segThree, segFour))
212
213             kOneRSME = self.ridgeRegressionBase(kOneData[:,1:],
214 kOneData[:,0], regTerm, segFive)
215
216             kTwoData = concatenate((segOne, segTwo, segThree, segFive))
217
218             kTwoRSME = self.ridgeRegressionBase(kTwoData[:,1:],
219 kTwoData[:,0], regTerm, segFour)
220
221             kThreeData = concatenate((segOne, segTwo, segFour, segFive))
222
223             kThreeRSME = self.ridgeRegressionBase(kThreeData[:, 1:],
224 kThreeData[:, 0], regTerm, segThree)
225
226             kFourData = concatenate((segOne, segThree, segFour, segFive))
227
228             kFourRSME = self.ridgeRegressionBase(kFourData[:, 1:],
229 kFourData[:, 0], regTerm, segTwo)
230
231             kFiveData = concatenate((segTwo, segThree, segFour, segFive))
232
233             kFiveRSME = self.ridgeRegressionBase(kFiveData[:, 1:],
234 kFiveData[:, 0], regTerm, segOne)
235
236             avgRSME = (kOneRSME + kTwoRSME + kThreeRSME + kFourRSME +
237 kFiveRSME) / 5
238
239             if minRSME is None:
240                 minRSME = avgRSME
241                 minRegTerm = regTerm
242             if avgRSME <= minRSME:
243                 minRSME = avgRSME
244                 minRegTerm = regTerm
245
246         regTerm /= 2
247
248         return minRegTerm
249
250 if __name__ == "__main__":

```

```
246     t = HW3()  
247  
248     t.linearRegression()  
249     t.linearRegressionGradientDescent()  
250     t.ridgeRegression()  
251     t.ridgeRegressionGradientDescent()  
252
```