Sean Morton
MDS Homework 4
10/11/21

"1. Write a code that can perform non-linear regression without any regularization, and with L1 and L2 regularization for the relationship shown below. Generate your training data with Gaussian noise and compare the performance with true data. Show how changing the penalty parameter can affect your prediction.

$$y = \frac{1}{4}e^{x} - \frac{1}{8}sin(x) - cos(x) + 0.1x^{3}$$

2. Perform linear and nonlinear regression using your dataset.
In order to learn more about the process of regression and to get some more regression practice, I made my own regression model, called "MortonRegression". It can perform linear regression and polynomial regressions of theoretically any degree, and can use L1 and L2 norm regularization.

L1 (Lasso) output of my model with lambda = 0.8:

```
Coeffs:  [0.2133742354996422, 0.5228026582628855, -0.00019774964336066118, -0.0003427192851529137]
Error:  3.192154158679316
R_squared:  0.9795333596944122
Iteration:  9000
```

L2 (Ridge) output of my model with lambda = 0.8:

```
Coeffs:  [0.22551733666110585, 0.5451973220835865, -0.22070193901554588, -0.38965505612564577]
Error:  2.860432075023012
R_squared:  0.9811740376264229
Iteration:  9000
```

I found that increasing the value of the "punishment parameter", lambda, improved the fit of the model only up to a certain point–after which, the quality of the model suffered. When lambda was small, the model was often overfitted; when lambda was too large, the weights didn't match the original shape of the curve at all. I found lambda = 0.8 to be the value that worked best in testing.

In testing, I found that increasing lambda from 0.8 to 1.8 caused R^2 to drop slightly.

L1 (Lasso) output of my model with lambda = 1.8:

```
Coeffs:  [0.19157289328760044, 0.4498846951175957, 0.00038717673789691196, -7.37717403805232e-05]
Error:  1.3918756893189168
R_squared:  0.9628924424552652
Iteration:  9000
```

L2 (Ridge) output of my model with lambda = 1.8:

```
Coeffs:  [0.19170924642791748, 0.46844573556923297, 0.03682314308374214, -0.16309250010563278]
Error:  1.2517266130286149
R_squared:  0.9657427640715652
Iteration:  9000
```

Procedure

The Lasso (L1 norm) model was very useful in that it successfully reduced the weights of some of the parameters to approx. 0, like the 0th order and 1st order terms, that weren't necessary to the fitting of the model. I didn't observe that the L2 norm model reduced the magnitude of any weights of the 3rd-order polynomial I used to approximate the given function for this problem.

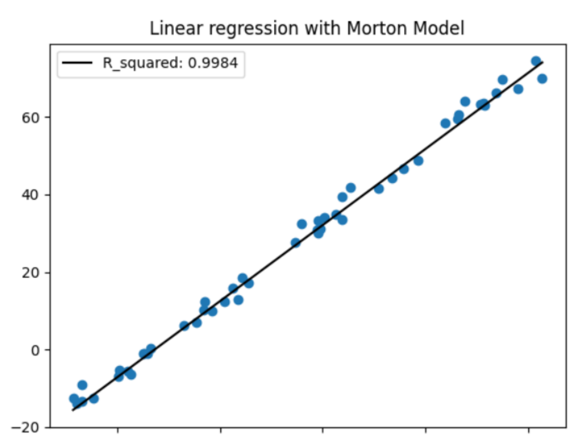
On the next page are some diagrams of the output of my model of a 3rd-order regression with lambda = 0.8 and 1.8. I added some gaussian noise to the y array, which could be tuned with the noise_stdev parameter. **The black curve represents the coefficients produced by my regression model, while the gray curve represents the original exponential/sine/power function** in the homework problem description.

Attached are the two Python files I used to carry out the regression. The first file contains the MortonRegression class, which sets up the polynomial regression and contains the member functions needed for regression. The second file uses the MortonRegression class to carry out no-norm, L1 norm, and L2 norm regularization regressions. I also compared my results to the results of the SKLearn linear model, Lasso model, and Ridge model to get a baseline for what the model output is supposed to look like.

Analysis

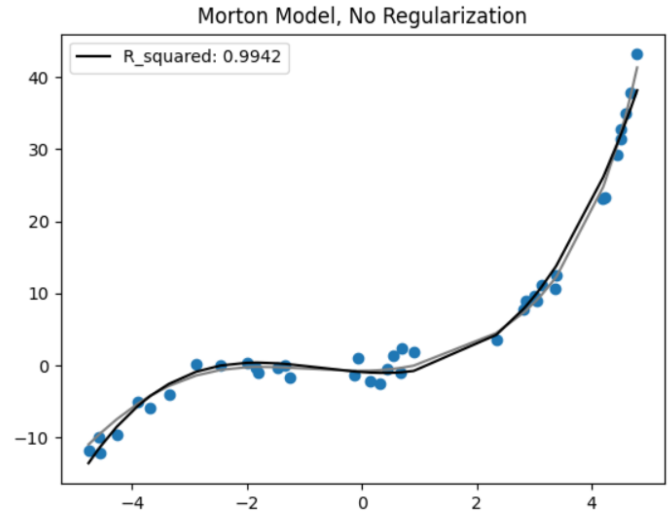
All three models, no-norm / L1 norm / L2 norm, approximated the original function fairly well within the bounds studied. The R^2 value of each of the $y_{predicted}$ arrays is above 0.99, and a side-by-side comparison shows that the black curve matches the gray curve very well in each of the graphs. The regression sometimes blows up to infinite values of the coefficients, but this is a shortcoming that could be improved in time and with different alpha. Overall this was a very effective way of studying regression and learning how to do it manually, rather than relying on SKLearn models.

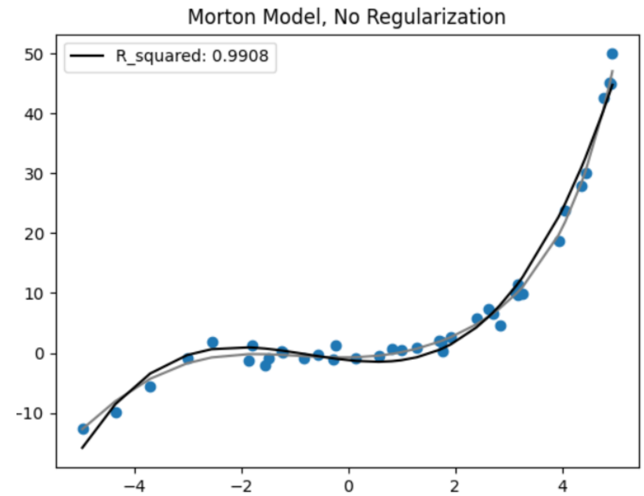
I could definitely improve this regression model in time–to make the model more efficient, or to make the predicted model match the original equation more.

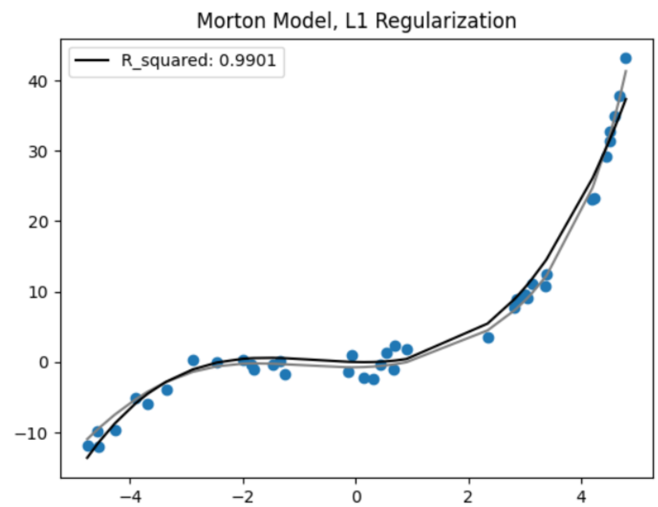


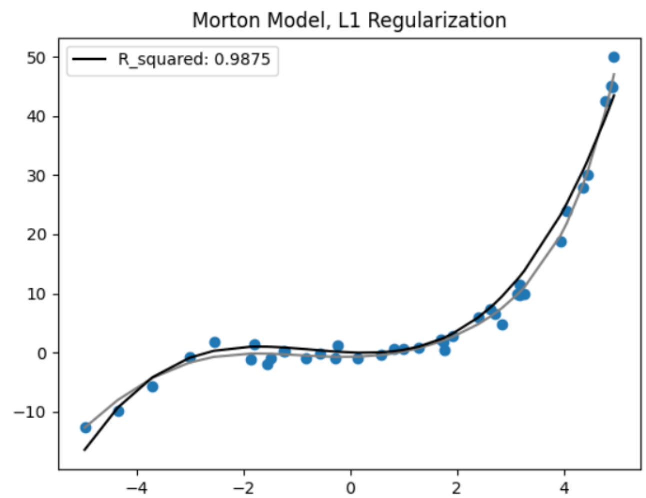
Here's an application of my model with y = 9.8x + 32.2; Gaussian noise of magnitude 2 applied

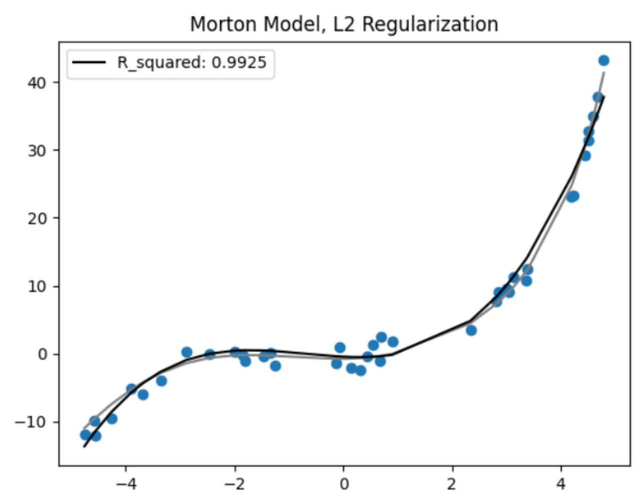
Models with Lambda = 0.8:	Models with Lambda = 1.8:

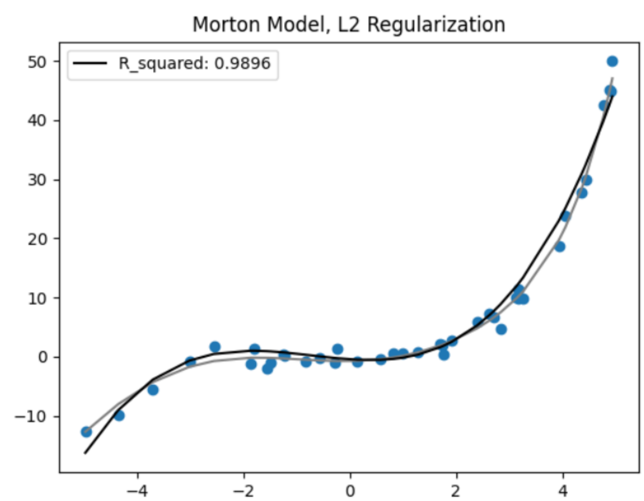












Code: general_nonlinear_regression.py

```
import numpy as np
import matplotlib.pyplot as plt
import math
import time
import random
import statistics
from scipy.stats import pearsonr

class MortonRegression():

    def __init__(self, degree=3):
        self.coeffs = [0.1] * (degree + 1)
        self.max_iters = 10000
        self.mse_threshold = 0.01

    def MSE(self, x_array, y_array):
        '''different values of omega will be the diff. coefficients in the array
        assume (n+1) coefficients for a nth-order polynomial

        assume length of "coeffs" array is unknown, but that
        coeffs corresponds to highest order down to lowest order
        this means where n = length of xoeffs array, coeffs has
        coefficients for x^(n-1) down to x^0'''

        num_values = len(x_array)
        if len(x_array) != len(y_array):
            raise ValueError("x and y arrays aren't the same size")

        #iterate through each value in the array
        mse_sum = 0
        for i in range(len(x_array)):
            x_n = x_array[i]
            y_n = y_array[i]
            y_pred = sum([self.coeffs[j] * x_n**(len(self.coeffs) - j - 1) for j in
range(len(self.coeffs))])
            mse_sum += (y_n - y_pred)**2

        return mse_sum / num_values

    #so the iteration happens here
    def derivatives(self, x_array, y_array, lambda_ = 0, L1 = False, L2 = False):
        '''iterate through all the derivatives in the nth-order list of coeffs
        -different values of omega will be the diff. coefficients in the array
        -assume 6 coefficients for a 5th-order polynomial
```

```
        -iterate through each value in the array'''

        num_values = len(x_array)
        if len(x_array) != len(y_array):
            raise ValueError("x and y arrays aren't the same size")

        deriv_array = []
        for n in range(len(self.coeffs)):

            xn_power = len(self.coeffs) - n - 1
            deriv_sum = 0
            for i in range(len(x_array)):
                x_n = x_array[i]
                y_n = y_array[i]
                y_pred = sum([self.coeffs[j] * x_n**(len(self.coeffs) - j - 1) for j
in range(len(self.coeffs))])
                deriv_sum += x_n**xn_power * (y_n - y_pred)

            nth_deriv = deriv_sum * (-2 / num_values)

            #implement L1 or L2 regularization
            if L1:
                if self.coeffs[n] >= 0:
                    nth_deriv += lambda_
                else:
                    nth_deriv -= lambda_

            elif L2:
                temp = lambda_ * self.coeffs[n] * (sum([x**2 for x in self.coeffs])
** -0.5)
                nth_deriv += temp

            deriv_array.append(nth_deriv)

        return deriv_array

    def train_model(self, x_array, y_array, alpha = 0.0003,
                    lambda_ = 0, L1= False, L2 = False):

        iter = 0
        error = 1000

        #iterate through each time and improve model
        while iter < self.max_iters and error > self.mse_threshold:

            #make our first guess as to the model. w0*x_n^5 + w1*x_n^4 + ... +
```

```
w5*x_n^0, e.g.
            y_pred = [sum([self.coeffs[j] * x_n**(len(self.coeffs) - j - 1) \
                for j in range(len(self.coeffs))]) for x_n in x_array]
            r = pearsonr(y_array, y_pred)[0]
            r_sq = r**2

            #find error of the current approximation
            error = self.MSE(x_array, y_array)
            derivs = self.derivatives(x_array, y_array, lambda_, L1, L2)

            #update coefficients based on gradient descent
            self.coeffs = [self.coeffs[ii] - alpha * derivs[ii] for ii in
range(len(self.coeffs))]

            if iter % 1000 == 0:
                print('\nCoeffs: ', self.coeffs)
                print('Error: ', error)
                print('R_squared: ', r_sq)
                print('Iteration: ', iter)
            iter += 1
            #time.sleep(0.5)

        #return coeffs

#main function; otherwise we just import the functions
#from this module
if __name__ == '__main__':

    #parameters for array
    num_values = 50
    x_scale_factor = 4
    x_offset = -1.5
    noise_stdev = 0.5

    #start with a vector of x
    x_array = x_scale_factor * np.random.rand(1, num_values)[0] + x_offset
    x_array.sort()

    #make y_array with actual values of first, second weight
    #y_array = 1.8*x_array**5 - 4.2*x_array**4 + 1.6*x_array**3 + x_array**2 \
    #        -5.2*x_array + 2.1
    y_array = 0.21*x_array**5 - 2.2*x_array**2 + 0.87

    #add some gaussian noise to the dataset
    noise = np.random.normal(0, noise_stdev, num_values)
    y_plus_noise = y_array + noise
```

```
#----------------------------------------#

#set up learning algorithm for regression.
reg = MortonRegression(degree=5)
alpha = 0.00003
lambda_ = 0.8
reg.train_model(x_array, y_array, L2 = True)
coeffs = reg.coeffs

#take our coefficients and show what y would equal
y_pred = [sum([coeffs[j] * x_n**(len(coeffs) - j - 1) \
        for j in range(len(coeffs))]) for x_n in x_array]

r = pearsonr(y_array, y_pred)[0]
r_sq = r**2

x_plot = np.arange(x_offset, x_offset + x_scale_factor, 0.01)
y_plot = [sum([coeffs[j] * x_n**(len(coeffs) - j - 1) \
        for j in range(len(coeffs))]) for x_n in x_plot]

plt.figure(1)
#plt.scatter(x_array, y_array)
plt.scatter(x_array, y_plus_noise)
plt.plot(x_plot, y_plot,
         label='R_squared: ' + str(round(r_sq, 2)))
plt.title('Sample nonlinear regression results')
plt.legend()
plt.show()
```

Code: hw4_nonlinear_regression.py

```
#typical scipy stuff
import numpy as np
import matplotlib.pyplot as plt
import math
import time
import random
from scipy.stats import pearsonr

from general_nonlinear_regression import MortonRegression

#sklearn module is helpful for machine learning
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression, Lasso, Ridge


num_values = 40
scale_factor = 10
offset = -5
noise_stdev = 1.2

#make a random array of x values
x_array = scale_factor * np.random.rand(1, num_values)[0] + offset
x_array.sort()
y_array = [1/4*math.exp(x) - 1/8*math.sin(x) - math.cos(x) + 0.1*x**3 for x in
x_array ]

#make a gaussian noise array and add it to y_array. let mean, stdev of
#gaussian distribution be the first inputs
noise = np.random.normal(0,noise_stdev,num_values)
y_plus_noise = y_array + noise

#-------------------------------------------------#
#part 1: pretrained model w/ no L1 norm regulatization

#preprocessing on polynomial regression sets up all the
#x variables to n degrees: 1, x, x^2, ..., x^n
pre_process = PolynomialFeatures(degree=3)

#transforms input to a vector that factors in 1, x, x^2, etc.
x_poly = pre_process.fit_transform(x_array.reshape(-1, 1) )

#fir the data to a model. this uses the LinearRegression()
#function but obv. it works with polynomials too
pr_model = LinearRegression()
```

```
pr_model.fit(x_poly, y_plus_noise)

#put predicted values of y created by model into an array
y_predicted = pr_model.predict(x_poly)

r_a = pearsonr(y_array, y_predicted)[0]
r_sq_sklinear = r_a**2

plt.figure(1)
plt.scatter(x_array, y_plus_noise)
plt.plot(x_array, y_predicted, c='black',
         label='R_squared: ' + str(round(r_sq_sklinear, 4)))
plt.title('SKLearn LinearRegression model')
plt.legend()
#----------------------------------------------------------#
#part 2: pretrained model with L1 norm regularization

lasso_model = Lasso()
lasso_model.fit(x_poly, y_plus_noise)

#put predicted values of y created by model into an array
y_lasso = lasso_model.predict(x_poly)
r_a1 = pearsonr(y_array, y_lasso)[0]
r_sq_sklasso = r_a1**2


plt.figure(2)
plt.scatter(x_array, y_plus_noise)
plt.plot(x_array, y_lasso, c='black',
         label='R_squared: ' + str(round(r_sq_sklasso, 4)))
plt.title('SKLearn Lasso Model')
plt.legend()



#-------------------------------------------------------#
#part 3: pretrained model with L2 norm regularization

ridge_model = Ridge()
ridge_model.fit(x_poly, y_plus_noise)

#put predicted values of y created by model into an array
y_ridge = ridge_model.predict(x_poly)
r_b = pearsonr(y_array, y_ridge)[0]
r_sq_skridge = r_b**2
```

```
plt.figure(3)
plt.scatter(x_array, y_plus_noise)
plt.plot(x_array, y_ridge, c='black',
         label='R_squared: ' + str(round(r_sq_skridge, 4)))
plt.title('SKLearn Ridge Model')
plt.legend()



#--------------------------------------------------------#
#part 4: the nonlinear regression model I built for practice

morton_model = MortonRegression(degree=3)
morton_model.train_model(x_array, y_plus_noise, lambda_ = 0.8)
coeffs = morton_model.coeffs

#take our coefficients and show what y would equal
y_morton = [sum([coeffs[j] * x_n**(len(coeffs) - j - 1) \
        for j in range(len(coeffs))]) for x_n in x_array]

r_c = pearsonr(y_array, y_morton)[0]
r_sq_morton = r_c**2

plt.figure(4)
plt.scatter(x_array, y_plus_noise)
plt.plot(x_array, y_array, c='gray')
plt.plot(x_array, y_morton, c='black',
         label='R_squared: ' + str(round(r_sq_morton, 4)))
plt.title('Morton Model, No Regularization')
plt.legend()

#plt.show()

#---------------------------------------------------------------#
#part 5: my nonlinear regression model with L1 reg.

morton_lasso = MortonRegression(degree=3)
morton_lasso.train_model(x_array, y_plus_noise, lambda_ = 1.8, L1 = True)
coeffs = morton_lasso.coeffs

#take our coefficients and show what y would equal
y_L1 = [sum([coeffs[j] * x_n**(len(coeffs) - j - 1) \
        for j in range(len(coeffs))]) for x_n in x_array]

r_d = pearsonr(y_array, y_L1)[0]
r_sq_L1 = r_d**2
```

```
plt.figure(5)
plt.scatter(x_array, y_plus_noise)
plt.plot(x_array, y_array, c='gray')
plt.plot(x_array, y_L1, c='black',
         label='R_squared: ' + str(round(r_sq_L1, 4)))
plt.title('Morton Model, L1 Regularization')
plt.legend()

#plt.show()

#----------------------------------------------------------#

#part 6: my nonlinear regression model with L2 reg.

morton_ridge = MortonRegression(degree=3)
morton_ridge.train_model(x_array, y_plus_noise, lambda_ = 1.8, L2 = True)
coeffs = morton_ridge.coeffs

#take our coefficients and show what y would equal
y_L2 = [sum([coeffs[j] * x_n**(len(coeffs) - j - 1) \
        for j in range(len(coeffs))]) for x_n in x_array]

r_e = pearsonr(y_array, y_L2)[0]
r_sq_L2 = r_e**2

plt.figure(6)
plt.scatter(x_array, y_plus_noise)
plt.plot(x_array, y_array, c='gray')
plt.plot(x_array, y_L2, c='black',
         label='R_squared: ' + str(round(r_sq_L2, 4)))
plt.title('Morton Model, L2 Regularization')
plt.legend()

plt.show()
```