Kenneth Le

UID:804953883

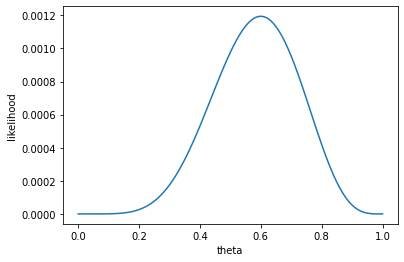
PS2

CS146

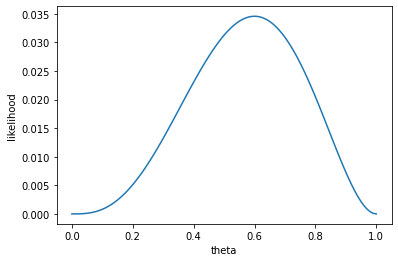
PS2 Write up

Question 3:

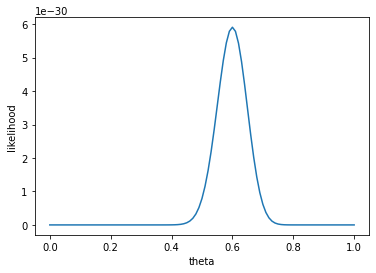
Part C.) For the MLE I stated that theta is equal to 1/n \* summation of Xi to n. Here we see from the graph that the MLE theta is 0.6. Our data consisted of 6 1s and 4 0s so the sum of this is 6. 6 divided by the number of samples 10 is equal to 0.6 which shows that the MLE closed form solution is correct.



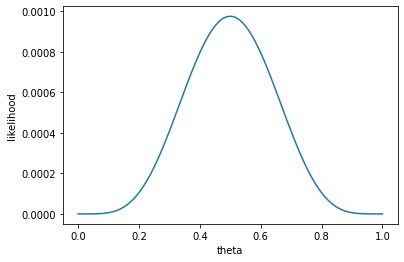
Part D.) For 3 1s and 2 0s we see that theta MLE is 0.6 again.



For 60 1s and 40 0s we get the same MLE again of 0.6.



For 5 1s and 5 0s we get an MLE of 0.5.



When observing these graphs it is clear to see that the graphs with the larger number of samples have peaks that are much larger and narrower than those of the ones that had less sample sizes even though the MLE was the same. A change in the MLE creates a shift in the graph as well from left to right depending on where the mean is situated.

Question 4

a.)

### ========== TODO : START ========== ###

# part a: main code for visualizations

print("Part A:")

print('Visualizing data...')

print("Train Data:")

plot\_data(X\_train,y\_train);

print("Test Data:")

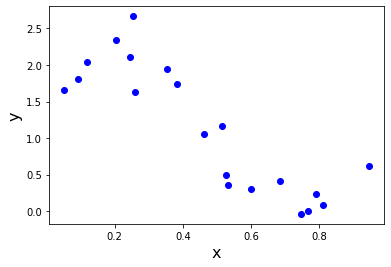
plot\_data(X\_test,y\_test,color = 'r');

### ========== TODO : END ========== ###

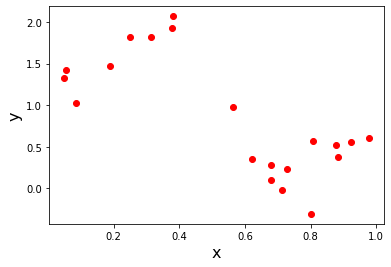
Part A:

Visualizing data...

Train Data:



Test Data:



Visualizing both the training data and the test data on the scatter plot I observed that the training data seems to be able to follow a general line from top left to bottom right which would work well with a linear classifier. For the test data the general direction is similar to the training data however there are not many points in the middle of the scatter plot while at the ends they are much more spread apart and perpendicular to the linear regression line that would be created from the training data which will influence the new model considerably.

b.)

# part b: modify to create matrix for simple linear model

matrix = np.ones([n,1])

X = np.append(matrix,X,1)

We create a matrix [1] and append X to it to get our X.

c.)

### ========== TODO : START ========== ###

# part c: predict y

y = np.dot(X,np.transpose(self.coef\_))

### ========== TODO : END ========== ###

This is just the dot product of the transpose of theta and X as h\_theta(X)= theta \* X

d.)

### ========== TODO : START ========== ###

# part d: compute J(theta)

cost = 0

h\_theta = self.predict(X)

for i,j in enumerate(h\_theta):

cost+= np.power((h\_theta[i] -y[i]),2)

print(cost)

### ========== TODO : END ========== ###

We create our function h\_theta using the predict function and then use enumerate to get tuples of the index and value to compute cost for each index i and add them together to get our total cost.

### ========== TODO : START ========== ###

# part d: update theta (self.coef\_) using one step of GD

# hint: you can write simultaneously update all theta using vector math

weight\_vec = np.array(list(self.coef\_))

for i, val in enumerate(self.coef\_):

summation = 0

for j, x in enumerate(X):

summation += (np.dot(weight\_vec, x)- y[j]) \* x[i] # [h\_theta(x\_n)-y\_n]\*x\_nj

self.coef\_[i] += -2 \* eta \* summation # -2\*n\*summation

# track error

# hint: you cannot use self.predict(...) to make the predictions

y\_pred = np.dot(X,np.transpose(self.coef\_)) # change this line

err\_list[t] = np.sum(np.power(y - y\_pred, 2)) / float(n)

### ========== TODO : END ========== ###

After implementing the gradient descent function fit\_GD I used the following step size values as described of 0.1,0.01,0.001,0.0001.

I got the following output:

Investigating linear regression...

part D: Gradient Descent

40.233847409671

number of iterations: 10000

4.086397036795765

eta: 0.0001 coefficients:[ 2.27044798 -2.46064834] cost:4.086397036795765

number of iterations: 7020

3.9125764057919463

eta: 0.001 coefficients:[ 2.4464068 -2.816353 ] cost:3.9125764057919463

number of iterations: 764

3.9125764057914862

eta: 0.01 coefficients:[ 2.44640703 -2.81635346] cost:3.9125764057914862

<ipython-input-30-cd69110fe7eb>:107: RuntimeWarning: overflow encountered in power

err\_list[t] = np.sum(np.power(y - y\_pred, 2)) / float(n)

<ipython-input-30-cd69110fe7eb>:111: RuntimeWarning: invalid value encountered in subtract

if t > 0 and abs(err\_list[t] - err\_list[t-1]) <= eps :

<ipython-input-30-cd69110fe7eb>:101: RuntimeWarning: overflow encountered in double\_scalars

summation += (np.dot(weight\_vec, x)- y[j]) \* x[i] # [h\_theta(x\_n)-y\_n]\*x\_nj

<ipython-input-30-cd69110fe7eb>:102: RuntimeWarning: invalid value encountered in double\_scalars

self.coef\_[i] += -2 \* eta \* summation

number of iterations: 10000

nan

eta: 0.1 coefficients:[nan nan] cost:nan

Investigating polynomial regression...

Done!

| Step size N | iterations |
| --- | --- |
| 0.0001 | 10000 |
| 0.001 | 7020 |
| 0.01 | 764 |
| 0.1 | NaN |

From this we can see that the smallest step value of 0.0001 took the most iterations for the convergence while 0.001 took 7020 iterations and 0.01 took 764 iterations which led to convergence the quickest.0.01 came out to a nan value and this is due to the fact that because the step size was too large and thus was unable to converge resulting in overflow. With respect to the coefficients, we see that 0.0001 step size differed from the other two other valid step sizes and was unable to fully converge after 10000 iterations hence the difference in coefficient values and as expected the cost is higher as a result.

e-f.)

### ========== TODO : START ========== ###

# part e: implement closed-form solution

# hint: use np.dot(...) and np.linalg.pinv(...)

# be sure to update self.coef\_ with your solution

Self.coef\_ = np.dot(np.dot(np.linalg.pinv(np.dot(np.transpose(X),X)),np.transpose(X)),y)

### ========== TODO : END ========== ###

When implementing the closed form solution and then using.fit() the following times were recorded.

print("part E:")

start = time.time()

model.fit(X\_train,y\_train)

stop = time.time()

runtime = stop - start

print(f'\tcoefficients:{model.coef\_} model\_fit closed form runtime:{runtime}')

start = time.time()

model.fit\_GD(X\_train,y\_train)

stop = time.time()

runtime = stop - start

print(f'\tcoefficients:{model.coef\_} model\_fit\_GD runtime:{runtime}')

Output:

part E:

coefficients:[ 2.44640709 -2.81635359] model\_fit runtime:0.0002753734588623047.

# part f: update step size

# change the default eta in the function signature to 'eta=None'

# and update the line below to your learning rate function

if eta\_input is None :

eta = 1/1+t # change this line

else :

eta = eta\_input

### ========== TODO : END ========== ###

For .fit\_GD() where n = 1/1+k I got the following information.

number of iterations: 764

coefficients:[ 2.44640703 -2.81635346] model\_fit\_GD runtime:0.11369061470031738

Here we can see that the closed form solution took an order of magnitude 10^-3 less time than when using gradient descent with the learning function and that gradient descent took 764 iterations for convergence.

g.)

### ========== TODO : START ========== ###

# part b: modify to create matrix for simple linear model

matrix = np.ones([n,1])

m = self.m\_

for i in range (1,m+1):

matrix = np.append(matrix,np.power(X,i),1)

Phi = matrix

### ========== TODO : END ========== ###

### ========== TODO : START ========== ###

# part h: compute RMSE

error = np.power(self.cost(X,y)/len(X),0.5)

### ========== TODO : END ========== ###

### ========== TODO : START ========== ###

# parts g-i: main code for polynomial regression

print('Investigating polynomial regression...')

train\_rmse\_error = []

test\_rmse\_error = []

x\_index = np.arange(11)

for m\_feature in range(11):

model = PolynomialRegression(m = m\_feature)

model.fit(X\_train,y\_train)

train\_rmse\_error.append(model.rms\_error(X\_train,y\_train))

test\_rmse\_error.append(model.rms\_error(X\_test,y\_test))

plt.plot(x\_index,train\_rmse\_error,color = "blue",label="RMS Training Error")

plt.plot(x\_index,test\_rmse\_error,color = "red",label="RMS Test Error")

plt.xlabel("Degree")

plt.ylabel("RMSE")

plt.title("Degree of Polynomial vs. Root Mean Squared Error")

plt.legend()

plt.show()

### ========== TODO : END ========== ###



One might find RMSE a better metric than J(theta) as it is a better estimation of the overall performance of a model while J(theta) is better at optimization when you are trying to search for the optimal parameters.

From the values of RMSE I found that out of the 11 degrees tested that when m = 5 we have the smallest training error and test error values and have the smallest difference between the two. With respect to underfitting we find that the degree of 0 suffered from underfitting as we see the training error is greater than that of the test error. From degrees 8-11 it suffered from overfitting as we see the test error jump considerably in difference from the training error which suggests that the model became increasingly more susceptible to noise/outliers in the data.

5

0.22681133051783178

0.3551377428803351