linreg

April 14, 2021

1 Assignment 04: Shallow ML - Linear Regression

In this assignment, you will implement the most famous shallow learning algorithm: the linear regressor. We will restrict ourselves to the univariate case, but the transfer to multiple variables is straight forward. Please also refer to the class notes for more details and derivations.

Some notes: - Each cell starts with a comment on its contents and function to help you understand what is going on - Cells in which your input is required have a ### STUDENT ### Tag in the first line - if a cell does not have such a tag, you do not need to change anything there - You can also feed your own data to the regressor, just give your file the name 'data.csv'. - If you have problems with plots not showing up, try adding either '%matplotlib inline' or '%matplotlib notebook' to the cell in question - this is called a magic function in ipython and usually helps with that

1.1 The task

Your task to apply a linear regressor to the problem of predicting which grade or number of points a student will receive on the final exam. The data may have been collected from field studies among LRT students in Stuttgart. The only input feature is the number of hours studied before the test, the only output is the points achieved. This calls for a linear regression model with one feature.

Your tasks are: - implementing the forward pass - computing the cost function - computing the gradients and updating the parameters - optimizing the model - comparing the results with the solution of the normal form of the least squares problem

Let us start by reading in the data and plotting it:

[ ]:
```
# Import bibs and helper functions
#########################################################################################

import numpy as np
from numpy import genfromtxt
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from IPython import display
import matplotlib.cm as cm
from matplotlib.colors import LogNorm
from matplotlib import rcParams
rcParams.update({'font.size': 24})
from IPython.display import Math, Latex, Markdown
```

```
################################################################################
# helper function, allows printing of markdown syntax
# z.B: printmd(**Fettschrift**)
def printmd(string):
    display.display(Markdown(string))


# Declare Variables
################################################################################
filename="data.csv"
```

```
[ ]: # Read Training Data and plot it
################################################################################
data_from_file = genfromtxt(filename, delimiter=',')
# the file data.csv contains two columns, separated by comma. The first column␣
 ↪contains the no. of hours studied,
# the second column the points on the test. Each row thus contains a␣
 ↪groundtruth pair (X,Y)

X=data_from_file[:,0]

# To underline that the Ys are the (T)ruth, we call the variable tY
tY=data_from_file[:,1]

nsamples=X.size
print("Number of read in samples: ",nsamples)

# We work with a linear model with one feature, so we will have two parameters
weights=np.zeros((2,1))
print("Current weights w0 and w1: ",weights[0],weights[1])
print("Dimension of parameter vector theta",weights.shape,"\n")

print("We extend the input vector by a column of ones to account for the bias␣
 ↪term")
print("Our linear model with a single input feature x_1 and parameters w_0 and␣
 ↪w_1 is given for a sample m as:")
display.display(Math(r'y^m=w_0+w_1\,x^m=w_0\cdot 1+w_1\,x^m=[1\,\,\, x^m]␣
 ↪\begin{bmatrix}w_0 \\ w_1 \\ \end{bmatrix}'))
X=np.c_[ np.ones(nsamples), X ]
X=np.array(X)
tY=np.array(tY)
tY=tY.reshape(nsamples,1)


# making sure that the dimensions are correct, note that each row corresponds␣
 ↪to a single sample
```

```
print("Shape of the inputs X",X.shape)
print("Shape of the outputs tY",tY.shape)


# Plot the training data
#########################################################################################

plt.figure(figsize=(20, 8))
area = np.pi*50
colors = np.random.rand(nsamples)
plt.scatter(X[:,1],tY,s=area, c=colors, alpha=0.9)
plt.xlabel('Hours studied')
plt.ylabel('Points on final exam')
display.display(plt.gcf())
display.clear_output(wait=True)
plt.grid(color='k', linestyle='--', linewidth=0.5)
plt.show()
```

Just by inspecting the data, we can see that a linear model might be a good idea, but it will certainly not be perfect. And yes, there are some students with negative points as well….

Let us try the linear regression for this!

1.1.1 Task 1:

The first step in the supervised learning method is to make a prediction based on the current parameters. Implement the forward pass through the linear model, that is the prediction of the model given some input x as a function of the current parameter vector w. Your function should work on multiple samples at a time, i.e. it should accept inputs of the (:,2) like we have seen for X.

```
[ ]: ### STUDENT ###
     def forwardpass(X,w):
         # YOUR CODE HERE
         raise NotImplementedError()
```

```
[ ]: # This cell tests your function "forwardpass" with the autograder
     Xtest=[[1, 3],[3. ,1]]
     thetatest=[2 ,1]
     np.testing.assert_array_equal([5.,7.],forwardpass(Xtest,thetatest))
```

Let us now use your forwardpass to plot the current model:

```
[ ]: # Compute the current prediction of the model for all training samples and plot
     #########################################################################################

     def showprediction(X,tY,w):
         printmd("## Prediction for w0 = "+str(w[0])+" **and w1 = **"+str(w[1])+"\n")
         plt.figure(figsize=(20, 8))
```

```
    plt.scatter(X[:,1],tY,s=area, c=colors, alpha=0.9)
    plt.xlabel('Hours studied')
    plt.ylabel('Points on final exam')
    plt.grid(color='k', linestyle='--', linewidth=0.5)
    Y=forwardpass(X,w)
    plt.plot(X[:,1],Y, c='orange')
    plt.show()
```

[ ]:
```
### STUDENT ###
# You can play around with parameters below
##########################################################################################

weights[0]=100
weights[1]=-1.0
showprediction(X,tY,weights)
```

1.1.2 Task 2:

Now that we can compute the prediction for our choice of parameters w, let us compute a norm to evaluate how good a given choice is. Below, implement the cost function for the batch gradient approach.

[ ]:
```
### STUDENT ###
# Compute the cost function for the batch gradient approach
##########################################################################################

def computecost(X,tY,w,nsamples):
    C_loc=0.0

    # YOUR CODE HERE
    raise NotImplementedError()
    return C
```

[ ]:
```
# This cell tests your function "computecost" with the autograder
```

[ ]:
```
# Helper function, evaluate the cost on a parameter grid
##########################################################################################

# split parameter space in 100x100 grid
w0_vals=np.linspace(-10, 10.0, num=100)
w1_vals=np.linspace(-1, 4.0, num=100)
C_vals = np.zeros((w0_vals.size, w1_vals.size));

# compute the costs for all nodes in 100x100 grid
for i in range (0,w0_vals.size):
    for j in range (0,w1_vals.size):
        t = (w0_vals[i] , w1_vals[j]);
        C_vals[i,j] = computecost(X, tY, t,nsamples);


%matplotlib notebook
fig = plt.figure(figsize=(14,10))
w0_pos, w1_pos = np.meshgrid(w0_vals, w1_vals)
ax = fig.add_subplot(1, 1, 1,projection='3d')
p = ax.plot_surface(w0_pos, w1_pos, C_vals.T, cmap=cm.plasma)
ax.set_xlabel('w0')
ax.set_ylabel('w1')
ax.set_zlabel('Costs C')
plt.show()


# Plot the 2D version of the costs
#########################################################################################

fig = plt.figure(figsize=(8,6))
ax1 = fig.add_subplot(1, 1, 1)
ax1.set_xlabel('w0')
ax1.set_ylabel('w1')
cp=ax1.contour(w0_vals, w1_vals, C_vals.T, levels=[60, 70,75, 80, 90, 100,125,␣
 ↪150,175,250],cmap=cm.plasma,norm = LogNorm())
plt.clabel(cp, inline=1, fontsize=10)
```

1.1.3 Task 3:

In the cell below, compute the mean gradient over all samples as shown in class, and update the parameters w from this. LR corresponds to the learning rate and is a real number >0.

[ ]:
```
### STUDENT ###
def compute_Gradient_and_update_parameters(X,w,tY,nsamples,LR):
    # YOUR CODE HERE
    raise NotImplementedError()
```

[ ]:
```
# This cell tests your function "compute_Gradient_and_update_parameters" with␣
 ↪the autograder
t_weights=np.array([[1.0],[1.0]])
t_y=np.array([[12],[13],[14]])
t_samples=t_y.size
t_X=np.array([[1, 1],[1 ,2],[1 ,3]])
t_LR=120
#compute_Gradient_and_update_parameters(t_X,t_weights,t_y,t_samples,t_LR)
np.testing.assert_array_equal([[1201.0],[2401.
 ↪0]],compute_Gradient_and_update_parameters(t_X,t_weights,t_y,t_samples,t_LR))
```

```
[ ]: # GradientDescent: Optimize the square error iteratively and plot what is␣
      ↪happening
     #########################################################################################
     def gradientDescent(X,tY,w,LR,niter):
         # 2D plot of error surface
         fig=plt.figure(figsize=(40, 20))
         fig, ax = plt.subplots(nrows=2, ncols=1,figsize=(15,15))
         ax[0].size=(12,12)
         ax[0].contour(w0_vals, w1_vals, C_vals.T, levels=[60, 70,75, 80, 90,␣
      ↪100,125, 150,175,250],cmap=cm.plasma,norm = LogNorm())
         # plot prediction
         N=nsamples
         area = np.pi*30
         colors = np.random.rand(N)
         ax[1].scatter(X[:,1],tY,s=area, c=colors, alpha=0.9)
         plt.xlabel('Hours studied')
         plt.ylabel('Points on final exam')

         # let us save the cost history
         C_history = np.zeros((niter, 1));

         for n in range(0,niter):

             compute_Gradient_and_update_parameters(X,w,tY,nsamples,LR)

             # save costs for cost history
             C_history[n] = computecost(X, tY, w,nsamples);
             # plot current prediction and contour plot
             if n%100 == 0:
                 ax[0].scatter(w[0],w[1])
                 ax[1].plot(X[:,1],np.dot(X,w),linewidth=1.0)
                 display.display(plt.gcf())
                 display.clear_output(wait=True)
         return(C_history)
```

```
[ ]: ### STUDENT ###
     # Do the training!
     #########################################################################################

     %matplotlib inline

     # select learning rate, initial weights and iterations.
     # You can play around with this!
     LR=0.015
     weights[0]=-8.0
     weights[1]=1.0
     n_iter=2000
```

```
# Now do the actual training and observe the plots
Cost_hist=gradientDescent(X,tY,weights,LR,n_iter)

print('The optimized weights are: ',weights[0],weights[1])
print ('The optimized costs are: ',Cost_hist[-1])
# Plot the cost history after training
cost_fig=plt.figure(figsize=(30, 15))
plt.xlabel('Iteration',fontsize=24)
plt.ylabel('Cost C',fontsize=24)
plt.grid(color='k', linestyle='--', linewidth=0.5)
plt.plot(Cost_hist, linestyle='-', linewidth=4.5)
plt.show()
```

The last of the three plots above shows what is called the *cost history*, which is just the costs as a function of the training iteration. For more complex algorithms and training, looking at the cost history can reveal if the learning is successful or if e.g. overfitting is likely.

For this convex cost function, the cost drops rapidly and the algorithm converges in very few steps, given that the learning rate is not too large.

1.1.4 Task 4:

Train the model as best as you can and provide the final weights and cost function value below:

```
[ ]: ### STUDENT ###
     # uncomment these lines, copy them below the <begin solution> tag and complete␣
      ↪them
     # w0=
     # w1=
     # cost =
     # YOUR CODE HERE
     raise NotImplementedError()
```

```
[ ]: # This cell tests results for w0, w1 and cost
```

We have thus seen how linear regression works in practice. Note all the steps discussed here (defining the model, computing the forward pass, computing the cost function and its derivatives, updating the weights and looping over the iterations) are also present almost any other supervised learning method, in particular in neural networks.

1.1.5 Task 5

Let us now check our results against the solution of the normal form of the linear least squares problem (see notes from class). In the box below, implement the normal equation to compute the optimal parameters directly from the matrices given by the training features X and the vector tY. You can use the numpy function for the inverse, numpy.linalg.inv if you want. Try it for yourself, but in case you run into too much trouble, a great resource is given here: https://towardsdatascience.com/performing-linear-regression-using-the-normal-equation-6372ed3c57

[ ]:
```
### STUDENT ###
def compute_parameters_from_normal_form(X,tY):
    # YOUR CODE HERE
    raise NotImplementedError()
```

[ ]:
```
t_y=np.array([[12],[13],[14]])
t_X=np.array([[1, 1],[1 ,2],[1 ,3]])
compute_parameters_from_normal_form(t_X,t_y)
np.testing.assert_almost_equal([[11.0],[1.
 ↪0]],compute_parameters_from_normal_form(t_X,t_y))
```

This concludes assignment 4. Let's summarize some important points: - The linear regression model is linear in the parameters, but can deal with an arbitrary number of features - the number of column in X just increases - Optimizing the parameters of the model is done in a supervised learning manner: We learn from the given data points to predict the model response (for all possible inputs) - To define what we mean by optimum, we need a cost function - a norm on the error of the current predictions - This cost function is of course a function of the parameters. Thus, we can compute the gradient - There are many optimization methods, gradient descent is just the most basic one - but it works nicely for convex problems - For the linear regression with square cost function, we can check our results against the normal equation of the linear least squares problem.