Lab 11

This week, we are focusing on gradient descent. Like last week, we wil be using the fake employee dataset with the goal of finding the best parameters for linear regression. The goals for this week are:

1. ~Reviewing grid search and the drawbacks of it~
2. ~Motivate the process for gradient descent~
3. Detail the steps of gradient descent
4. Define the learning rate for gradient descent and the impact of it on the speed of the algorithm
5. Implement different versions of gradient descent

Imports for Today

Let us import the packages that we need for today and the dataset from last time.

In [ ]:
```
## Import block
%matplotlib inline

import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
import seaborn as sns; sns.set()

import numpy as np
from numpy import linalg as LA
import pandas as pd

from sklearn import linear_model
```

In [ ]:
```
## Functions for later use

def draw_vector(v0, v1, ax=None):
    ax = ax or plt.gca()
    arrowprops=dict(arrowstyle='->',
                    linewidth=2,
                    shrinkA=0, shrinkB=0, color="lightblue")
    ax.annotate('', v1, v0, arrowprops=arrowprops)

def place_parameter(p_vec, col, ax=None):
    plt.scatter(p_vec[0],p_vec[1], c=col, marker = "*", s = 100)

def draw_parameter_path(p0,p1, col, ax=None):
    ax = ax or plt.gca()
    arrowprops=dict(arrowstyle='->',
                    linewidth=2,
                    shrinkA=0, shrinkB=0, color=col)
    ax.annotate('', p1, p0, arrowprops=arrowprops)

def compute_mse(truth_vec, predict_vec):
    return np.mean((truth_vec - predict_vec)**2)
```

```
def compute_m_partial(in_vals, truth_vec, predict_vec):
    return -2*np.mean(in_vals*(truth_vec-predict_vec))

def compute_b_partial(truth_vec, predict_vec):
    return -2*np.mean(truth_vec-predict_vec)

def adjust_L(current_L, grad_step_num):
    pass
```

In [ ]:
```
## Import Data

employ_data = pd.read_csv("../Lab09/lab9data.csv", sep = ",")

## numpy vectors of our inputs
neuro = employ_data[["neuroticism"]].to_numpy()
perform = employ_data[["performance"]].to_numpy()
```

In [ ]:
```
# For function testing
```

Where are we?

Last class, we spent considerable time motivating the role of the gradient descent in stepping towards the minimum. As many of you noticed, our first 500 steps made some progress towards that goal.

This lab will be a periods of coding punctuated by discussion and brainstorming.

To begin, copy your code that creates the list of m values, b values, and the MSE associated to each pairing. I've copied the starter code from last lab to help you hone in on which block I mean:

In [ ]:
```
# Initialize the number of steps you wish to take
# Set your learning rate
n_steps = ???
L = 0.01

# Initialize starting parameters
m = ??
b = ??

# Create empty lists to store values for m, b, and the associated MSE
outm = []
outb = ???
outmse = ???

# Create an iterative process (ie. a loop) that will take N_STEPS
for stp in range(??):
    # For the current values of m and b:

    # 1. Compute the MSE
    preds = ???
    errormse = compute_mse(??,??)
```

```
    # 2. Store m, b, and the associated MSE in the output lists:
    outm.append(m)
    outb.append(??)
    outmse.??

    # Update m and b by:

    # 1. Computing the gradient
    d_m = compute_m_partial(???,???,???)
    d_b = ???

    # Update the values for m and b
    m = m - (?*d_m)
    b = ??
```

Exploring L

Fix the value for `n_steps` to **200.** This time, run gradient descent 3 times, each with a different value for L. Limit your options to between 0.00001 to 0.1. Plot the resulting paths again in comparison to the parameters selected by the `sklearn` implementation of linear regression.

What surprises you? What ideas/questions/concerns do you have?

In [ ]:
```
# Block for exploration
```

In [ ]:
```
# Block for additional exploration
```

In [ ]:
```
# Block for more exploration
```

In [ ]:
```
# Block for testing an idea (or two) for Gradient Descent
```

Brainstorming session

What do you notice? With your group, try to come up with at least 5 ideas/questions:

-
-
-
-
-

Considerations in Gradient Descent

We have two big considerations in gradient descent: 1) the size of the step that we are taking, and 2) the number of steps that we are taking.

The first is governed in part by the **learning rate** which we have denoted as L. The learning rate effectively controls how much of an effect the gradient has on the parameter update. The second -- like the number of iterations within k-means -- is a bit more subtle, requiring an examination of **stopping conditions.**

Learning Rate + Step Size

There are many ways to approach the learning rate and step size:

1. Enforce a consistent step size by shrinking the gradient vector to be of length one and keeping a consistent learning rate.
2. We could take an adaptative approach that is related to how many steps the algorithm has taken. That is, gradient descent could take bigger steps at the beginning of the path and then take increasingly smaller ones as it moves on.

While easy to state the intuition of this second option, there are many ways that one could do this:

- Decrease the value of L with each step of the gradient descent either linearly or exponetially
- Decrease the value of L in a stepwise fashion every few steps

Choose one of these approaches to the learning rate and sketch out how you would like to code this. Then with your group, implement one approach for varying L as the helper function `adjust_L` in the function block.

Aside: This page develops and expands the above ideas.

Adjusting L in Gradient Descent

Now let's incorporate your helper function into your version of gradient descent. I've copied an edited version of the above starter code below.

Apply this implementation to our employee information. Then chart the path your gradient descent took and again compare it to the parameter values found by the `sklearn` implementation of linear regression.

In [ ]:
```
# Scratch block
```

In [ ]:
```
# Gradient Descent with varying L

# Initialize the number of steps you wish to take
n_steps = ???

# Initialize starting parameters
m = ??
b = ??

# Initialize L
```

```
L = ???

# Create empty lists to store values for m, b, and the associated MSE
outm = []
outb = ???
outmse = ???

# Create an iterative process (ie. a loop) that will take N_STEPS
for stp in range(??):
    # For the current values of m and b:

    # 1. Compute the MSE
    preds = ???
    errormse = compute_mse(??,??)

    # 2. Store m, b, and the associated MSE in the output lists:
    outm.append(m)
    outb.append(??)
    outmse.??

    # Update m and b by:

    # 1. Computing the gradient
    d_m = compute_m_partial(???,???,???)
    d_b = ???

    # Update the values for m and b
    m = m - (?*d_m)
    b = ??

    # Update learning rate
    L = ???
```

In [ ]:

```
# m and b as given by sklearn:

# 1. Define the model:
sk_line = linear_model.LinearRegression()

# 2. Fit the model to our data:
sk_mod = sk_line.fit(neuro, perform)

# 3. Extract coefficients:
m_sk = sk_mod.coef_
b_sk = sk_mod.intercept_
```

In [ ]:

```
# Your plot of the path given by Gradient Descent
#                       with varying learning rate
```

Number of steps (or GD Stopping conditions)

Gradient descent at its core is a series of steps, but we have to say when we've taken enough steps. There are two considerations when determining that number. From a computational point

of view, we also want to limit the maximum number of steps. However, we want to take enough steps that we reach the minimum, but not so many that we by-pass that minimum.

In terms of testing to see if we have reached the minimum, let us recall a bit of calculus. Consider a one-dimensional curve, ie. one that you can draw on paper with your pencil. Draw x and y axes for reference.

Now find a minima, and lay your pencil down such that the side of the pencil (ie. neither "end" of the pencil) touches the curve in one place. Notice that your pencil is parallel to the x axis, or as we say, it's *flat*. Your pencil is a physical representative of the derivative. If we were to do something similar for an evaluation surface, we would use a board in place of the pencil. In this higher dimensional version, the board will be parallel to the parameter plane.

When the derivative flattens, the rate of changes are close to 0. So encoding this arrival at a "minima" is equivalent to checking the size of the gradient, or checking to see when the length of the gradient is close to zero.

We balance these consideration with two stopping conditions:

1. We can set a hard limit on how many steps we want to take
2. When the (length of the) gradient becomes small enough

These stopping conditions feel very similar to those we encountered for k-means, in that the first limits the number of iterations and the second does a quick examination of the most recent progress that the algorithm has made.

We will add these stopping conditions in two phases, checking the paths against the parameter values found by the `sklearn` implementation of linear regression.

Note: To compute the length of a vector, you can either use:

- `norm` within the linear algebra submodule (`linalg`) of `numpy`
- The fact that $|\nabla f| = \sqrt{\nabla f \cdot \nabla f}$, where ∇f denotes the gradient of the function f. In this case $\nabla f = [\text{partial with respect to m}, \text{partial with respect to b}]$.

We add these stopping conditions in waves. Again, I'll be copying previous code shells to give you a starting point

In [ ]:

```
# Adding a hard limit (ie. the max number of steps)

# Gradient Descent with varying L and a max-number of steps

# Initialize the max number of steps you wish to take
max_steps = ???

# Initialize starting parameters
m = ??
b = ??

# Initialize L
```

```
L = ???

# Create empty lists to store values for m, b, and the associated MSE
outm = []
outb = ???
outmse = ???

# Create an iterative process (ie. a loop) that will take N_STEPS
for stp in range(??):
    # For the current values of m and b:

    # 1. Compute the MSE
    preds = ???
    errormse = compute_mse(??,??)

    # 2. Store m, b, and the associated MSE in the output lists:
    outm.append(m)
    outb.append(??)
    outmse.??

    # Update m and b by:

    # 1. Computing the gradient
    d_m = compute_m_partial(???,???,???)
    d_b = ???

    # Update the values for m and b
    m = m - (?*d_m)
    b = ??

    # Update learning rate
    L = ???
```

Did anything substantively change in your implementation above?

In [ ]:

```
# Your plot of the path given by Gradient Descent with a maximum number of steps
```

In [ ]:

```
# Adding a stopping condition based on the length of the gradient
# Adding a hard limit (ie. the max number of steps)

# Gradient Descent with varying L and a max-number of steps

# Initialize the max number of steps you wish to take
max_steps = ???

# Initialize starting parameters
m = ??
b = ??

# Initialize L
L = ???

# Set a tolerance for the smallest you will allow
# the length of the gradient to be before stopping
grad_tol = 0.01
```

```
# Create empty lists to store values for m, b, and the associated MSE
outm = []
outb = ???
outmse = ???

# Create an iterative process (ie. a loop) that will take N_STEPS
for stp in range(??):
    # For the current values of m and b:

    # 1. Compute the MSE
    preds = ???
    errormse = compute_mse(??,??)

    # 2. Store m, b, and the associated MSE in the output lists:
    outm.append(m)
    outb.append(??)
    outmse.??

    # Update m and b by:

    # 1. Computing the gradient
    d_m = compute_m_partial(???,???,???)
    d_b = ???

    # Compute the length of the gradient
    norm_grad = ??

    # If the length of the gradient is small enough, stop iterating
    if norm_grad < ???:
        break

    # Update the values for m and b
    m = m - (?*d_m)
    b = ??

    # Update learning rate
    L = ???
```

In [ ]:

```
# Your plot of the path given by Gradient Descent with:
#        1. a maximum number of steps AND
#        2. length of the gradient
```

Gradient descent

Let us put all these pieces and implement gradient descent for linear regression with two stopping conditions and an option to vary the learning rate (or not!). Your implementation should be a function that takes in the following arguments:

1. Your input variable for regression
2. Your output variable for regression
3. The max number of steps
4. A flag allowing for one to vary (or not) the learning rate

Your implementation should **return** just the the values for m and b that gradient descent last computes before being stopped by a stopping condition

Apply your implementation to our employee information. Then chart the path your gradient descent algorithm took and again compare it to the parameter values found by the `sklearn` implementation of linear regression.

In [ ]:
```
# Scratch block
```

In [ ]:
```
# Your implementation of Gradient Descent as a function
```

In [ ]:
```
# Your plot of the path given by Gradient Descent
```

Different flavors of Gradient Descent

Last time we talked about the number of MSE computations for grid search. Clearly gradient descent lowers the number of MSE computations, but within each MSE computation, there are also a number of computations dictated by the size of the dataset.

Our dataset has 1000 employees. For each computation of MSE for our example, how many arithmatic operations occur? Think about the *order* of operations as you do this. Try to express your number in terms of n_o the number of observations, n_v the number of variables, and/or the n_p the number of parameters.

When you have a number, consult with at least one member of your group. (You may decide to use the chat function in gather or slack to do this.)

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) attempts to overcome the number of computations within the MSE by using just one randomly chosen datapoint for each step of the gradient descent. The gradient descent that we implemented earlier is also called *batch gradient descent* because it consults a *batch* of data, instead of just one datapoint, in the decision of where to step.

Create an implementation of stochastic gradient descent, where for each step, you randomly select one datapoint to act in place of the dataset. This means that you are *only using one* data point for computing **both** MSE and for computing the gradient. Before you get started, a few hints and warnings:

- You might want to start with copying your code from above
- Be careful to **not overwrite** the dataset when you select your random point
- You may want to see how lab 8 uses `np.random.choice()` to select 4 points in our second regression line.

Apply your implementation to our employee information. Then chart the path your stochastic gradient descent took in one color and the path your batch gradient descent took in a second color. Compare both paths to the parameter values found by the `sklearn` implementation of linear regression.

Aside: In the most precise version of SGD, we use each datapoint exactly once before repeating any of the data. Each pass over full dataset is called an *epoch*.

In [ ]:
```
# Scratch Block
```

In [ ]:
```
# Your implementation of Stochastic Gradient Descent
```

In [ ]:
```
# Your plot of the paths given by Stochastic and (Full) Gradient Descents
```

Mini-Batch Gradient Descent

Mini-Batch Gradient Descent is the compromise between batch and stochastic gradient descent using small collections of the data in each step, instead of using all of the data or using just one data point. Mini-batch takes *batches* of n_b datapoints in each step of the gradient descent.

Create an implementation of mini-batch gradient descent, where you set the size of the batches (that is the value of n_b), and where for each step, you randomly select n_b datapoint to act in place of the dataset. Before you get started, a few hints and warnings:

- You might want to start with copying your code from above
- Be careful to **not overwrite** the dataset when you select your random point
- You may want to see how lab 8 uses `np.random.choice()` to select 4 points in our second regression line.

Apply your implementation to our employee information. Then chart the path your mini-batch gradient descent took in one color and the path your batch gradient descent took in a second color. Compare both paths to the parameter values found by the `sklearn` implementation of linear regression.

Aside: In the most precise version of Mini-batch, we parcel the data into batches, and then use each batch exactly once before repeating any of the batches. Just as with SGD, each pass over full dataset is called an *epoch*.

In [ ]:
```
# Scratch Block
```

In [ ]:
```
# Your implementation of Mini-batch Gradient Descent
```

In [ ]:
```
# Your plot of the paths given by Mini-Batch Gradient Descents
```

Next week

A deeper look at the training and testing phases.

Final Thoughts

To finish up this lab, create a plot with both your stochastic and mini-batch gradient descent paths compared to the best values of m and b according to the linear regression from `sklearn`. Share your plot in a post on **#lab11_submission** channel on slack and note which path seems better to you.

If your have questions from this lab, post them to #lab_questions with the same preamble (i.e. starting with **Lab11**). If you have the same question, please use one of the emoji's to upvote the question. If you would like to answer someone's question, please use the thread function. This will tie your answer to their question.

Resources consulted

1. *Doing Data Science: Straight talk from the frontline* by C. O'Neil & R. Schutt (2014)
2. BATCH GRADIENT DESCENT VS STOCHASTIC GRADIENT DESCENT
3. Gradient descent algorithms and adaptive learning rate adjustment methods
4. Learning Rate Schedules and Adaptive Learning Rate Methods for Deep Learning
5. Stochastic Gradient Descent on Wikipedia
6. Gradient Descent Lecture notes by Ryan Tibshirani
7. norm helpfile in numpy
8. Notation for grad