Adaline

September 18, 2020

[1]:
```
'''Load this code, you don't have to change anything'''

import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets

def predict(X, w):
    a = X.dot(w) < 0
    a = a.astype(int)  # will be True if less than zero
    a[a == 1] = -1     # hence gets the label -1
    a[a == 0] = 1      # else gets the label 1 b/c greater than 0
    return a

def accuracy(X, w, y):
    return np.sum(predict(X, w) == y)/y.size

def loss(weights, features, labels):
    return np.sum((labels - features.dot(weights)) ** 2)

def grad(weights, features, labels):
    a =  -2 * np.multiply(features, (labels - features.dot(weights)))
    return np.sum(a, axis=0).reshape(-1, 1)
```

[58]:
```
# Load the iris dataset

iris = datasets.load_iris()
N = 100
X = iris.data[:, :2][0:N]  # we only take the first two features and 100 rows
y = iris.target[0:N]       # we only take the first two features and 100 rows

N, F = X.shape

y = y.reshape(N,1)

y[y == 0] = -1

# Initialize the weights
```

```
w = np.random.rand(F).reshape(F,1)
```

0.0.1 Building intuition for the loss function

Describe the loss function for Adeline in your own words. What does the equation say?

The loss function is the sum of the squared difference of the target variable (labels) and the net input to the output unit (features.dot(weights)). This function returns the SSE, which we want to minimize.

0.0.2 Examining the existing code

Using the slides on Adeline as a guide, please examine the `predict`, `accuracy`, `loss` and `grad` functions above. Please describe in your own words what each function is doing

The predict function takes the features dotted with the weights to create a bipolar class label based on the value of the dot product and returns a vector of predicted classifications for each observation.

The accuracy function sums the number of correctly classified observations from the predict function and divides that sum by the total number of observation to obtain a percentage of correct classifications.

The grad function calculates the gradient of the cost function and returns a scalar sum of each of the gradient components across all weights.

0.0.3 Optimization: Random search

In a minute we will optimize Adeline via a gradient-based technique. But first, let's take a minute to understand *why* we even want to do this and *what* we are even doing in the first place. We can also optimize the method via random search. To be clear, this is a bad way to actually optimize the function. But it is a great way to build intution for what is actually happening in gradient descent.

Start off by implementing the `get_random_weights` function below. The function should return weights returned at random.

[3]:
```
def get_random_weights(NumFeatures=2):
    w_ = np.random.rand(NumFeatures).reshape(NumFeatures,1)
    return w_
    '''

    Return a vector of shape NumFeatures X 1, filled at random. Numpy has␣
 ↪functions for this

    For this assignment NumFeatures=2
    '''
    pass
```

[4]:
```
def random_search(features, labels, iters=100):
    '''
    Implement this random search function
    Pseudocode is provided for you. Random search
    just keeps trying different weights at random
    and then returns the best weights it has found so far
    '''

    # initialize current_w and best_loss
    current_w = get_random_weights(2)
    best_loss = 10000

    for i in range(iters):
        new_w = get_random_weights(2)
        new_loss = loss(new_w, features, labels)


        # Implement the following...
        # If the new_loss is less than the best_loss
        # Then replace current_w with new_w and replace best_loss with the new␣
↪loss

        if(new_loss < best_loss):
            current_w = new_w
            best_loss = new_loss


    # return the best weights you found, during your random search
    return current_w, new_loss


random_search(features=X, labels=y)
```

[4]:
```
(array([[0.05411879],
        [0.07334382]]),
 714.5210647031687)
```

0.0.4 Optimization: Gradient descent

Random search is not a great way to optimize most functions, especially if you know the gradient which always points in the direction of greatest increase of a function. We will optimize Adeline with gradient *descent*, which takes steps in the *opposite* direction of the gradient. Why does that make sense? Hint: you should describe the Adeline loss function?

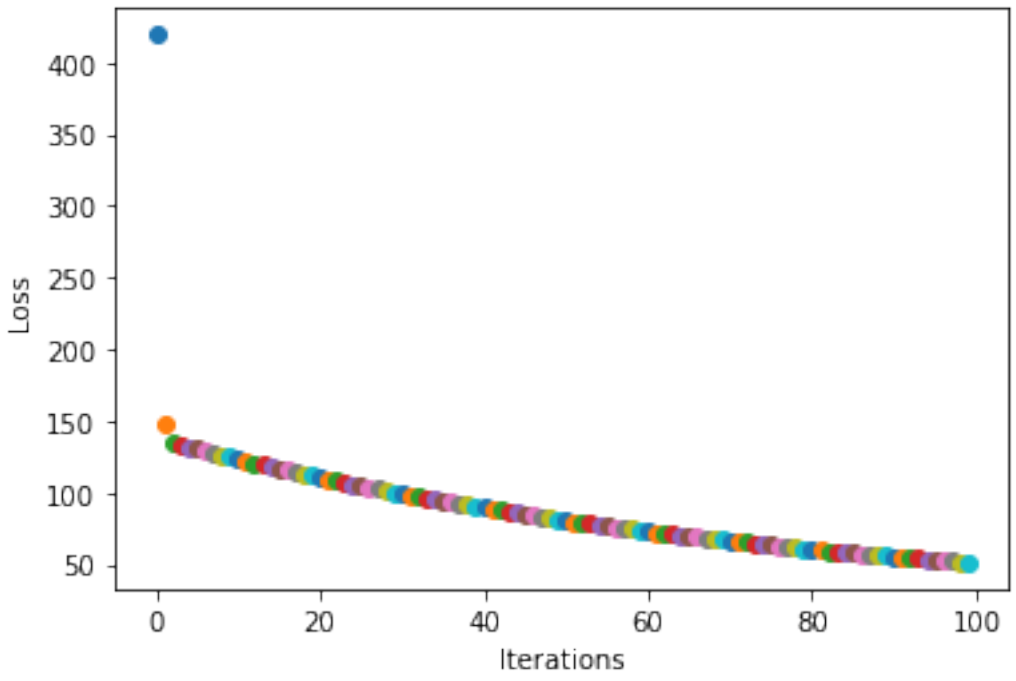
The gradient will always point in the direction of greatest increase. In this case the Adaline loss function is convex which means that the direction of greatest increase always is up and away from the min. Since we want to minimize the loss function, we move in the opposite direction of the gradient towards the min.

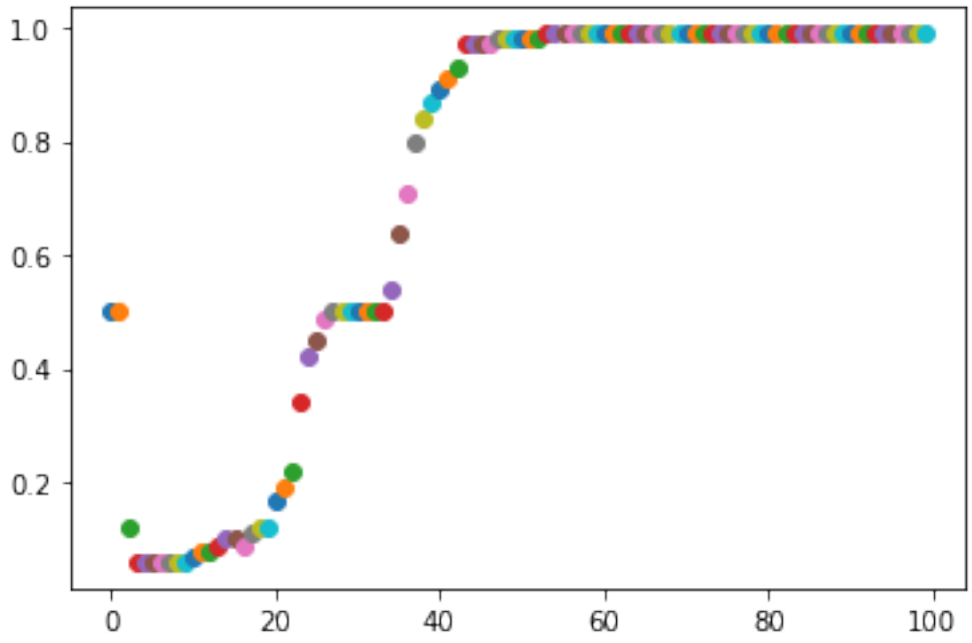
[59]:
```
ETA = .0001
ITERS = 100

# Using the slide on Gradient descent as a guide (along with the pseudocode␣
↪below)
# finish the implementation of gradient descent below


plt1 = plt.ylabel("Loss")
plt1 = plt.xlabel("Iterations")


for i in range(ITERS):

    old_loss = loss(weights=w, features=X, labels=y)

    ## implement the gradient descent step here
    w += - ETA * grad(w, X, y)

    new_loss = loss(weights=w, features=X, labels=y)
    acc = accuracy(X= X, w= w, y=y)
    plt1 = plt.figure(1)

    plt1 = plt.scatter(i, new_loss)
    plt2 = plt.figure(2)
    plt2 = plt.scatter(i, acc)

    ## the new_loss should be smaller than the old_loss, if you used eta = .0001

    print(loss(weights=w, features=X, labels=y), acc) # this should go down␣
↪each iteration
```

```
419.42312217743176 0.5
148.15850346906217 0.5
135.50406151190953 0.12
133.53409347464122 0.06
132.02348968683575 0.06
130.55030780135104 0.06
129.09690737345247 0.06
127.66232512260063 0.06
126.24628860959285 0.06
124.84855689486987 0.06
123.4688933072916 0.07
122.10706428269287 0.08
120.76283927809608 0.08
119.43599073077543 0.09
118.12629401965435 0.1
```

116.83352742727641 0.1
115.55747210227072 0.09
114.29791202230281 0.11
113.05463395750446 0.12
111.82742743437605 0.12
110.61608470015574 0.17
109.42040068764926 0.19
108.2401729805142 0.22
107.07520177899302 0.34
105.92528986608926 0.42
104.79024257418081 0.45
103.6698677520647 0.49
102.56397573242812 0.5
101.47237929973967 0.5
100.39489365855586 0.5
99.33133640223737 0.5
98.28152748206925 0.5
97.24528917678097 0.5
96.22244606246001 0.5
95.21282498285443 0.54
94.21625502005942 0.64
93.23256746558282 0.71
92.26159579178454 0.8
91.3031756236853 0.84
90.35714471113958 0.87
89.42334290136863 0.89
88.50161211184823 0.91
87.59179630354694 0.93
86.69374145451059 0.97
85.80729553378787 0.97
84.93230847569322 0.97
84.0686321544024 0.97
83.21612035887608 0.98
82.37462876810834 0.98
81.54401492669439 0.98
80.72413822071442 0.98
79.91485985392947 0.98
79.11604282428426 0.98
78.32755190071468 0.99
77.54925360025439 0.99
76.7810161654376 0.99
76.02270954199413 0.99
75.2742053568324 0.99
74.53537689630745 0.99
73.80609908476963 0.99
73.08624846339058 0.99
72.37570316926308 0.99
71.67434291477095 0.99

70.98204896722561 0.99
70.29870412876606 0.99
69.62419271651834 0.99
68.95840054301196 0.99
68.30121489684886 0.99
67.65252452362289 0.99
67.01221960708544 0.99
66.38019175055491 0.99
65.75633395856619 0.99
65.14054061875747 0.99
64.53270748399139 0.99
63.93273165470698 0.99
63.340511561499866 0.99
62.75594694792781 0.99
62.17893885353822 0.99
61.6093895971153 0.99
61.04720276014372 0.99
60.49228317048592 0.99
59.94453688627053 0.99
59.40387117998903 0.99
58.87019452279804 0.99
58.34341656902438 0.99
57.823448140870575 0.99
57.31020121331808 0.99
56.80358889922551 0.99
56.30352543461958 0.99
55.80992616417634 0.99
55.322707526889715 0.99
54.841787041925755 0.99
54.36708329465939 0.99
53.898515922891846 0.99
53.43600560324626 0.99
52.97947403773893 0.99
52.52884394052454 0.99
52.0840390248123 0.99
51.64498398995156 0.99
51.21160450868412 0.99





0.0.5 Loss by iteration

Modify the gradient descent code to plot the loss at each iteration of the algorithm. What do you observe?

As the iterations increase, the loss decreases.

0.0.6 Random search vs. gradient descent

Examine the value of the loss after 100 iterations of random search and 100 iterations of gradient descent. Which one optimizes faster? Does that make sense?

The gradient descent optimizes faster because it is actually adjusting the weights based on the learning rate multipled by the negative gradient. Random search is simply comparing 100 iterations of random weights wich may very far off from the optimal weights.

0.0.7 Accuracy by loss

Modify the gradient descent code to use the `accuracy` function to measure the accuracy at each iteration of the algorithm. Plot the relationship between accuracy and loss. What do you observe? Does that make sense?

This makes sense because the weights are being continually updated to better predict the class labels of the data. The loss is also decreasing as the number of interations increases. This means that Adaline is doing a better job of predicting the labels.

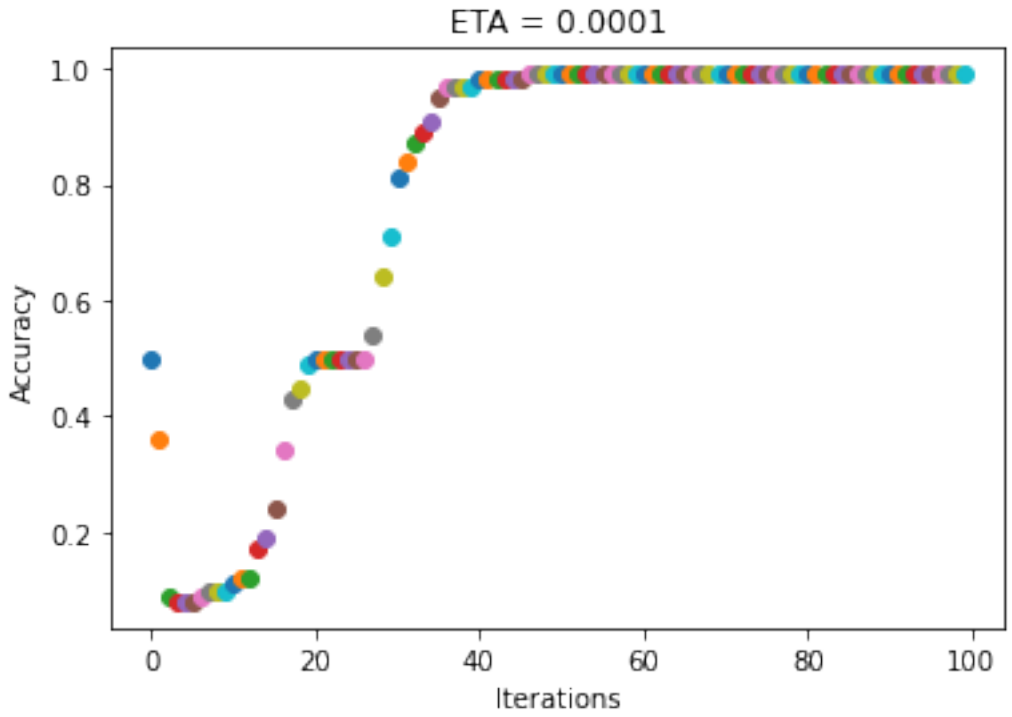
0.0.8 Learning rates

Try varing the learning rate eta by increasing or decreasing eta by powers of 10. Make a plot showing the learning rate and the accuracy after 100 iterations, for different valuse of eta. Does the algorithm achieve high accuracy for all eta, or only for some learning rates? Why might this be the case?

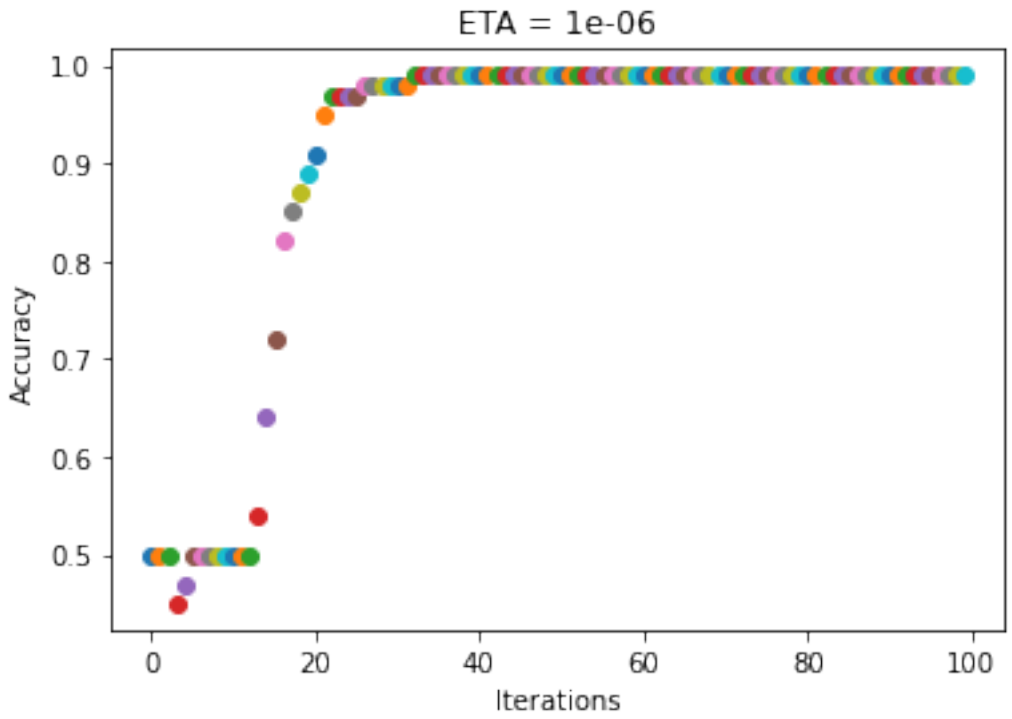
In the cases I tested, the algorithm eventually achieved high accuracy in every case. Theoretically, if an eta is too large the alorithm will overshoot the min, but since the loss function is convex, the negative gradient always points in the direction of the min. Generally, the algorithm should reach the min after enough iterations.

[64]:
```
def GD(eta, ITERS = 100, X = X, y= y):
    w = np.random.rand(F).reshape(F,1)
    plt2 = plt.ylabel("Accuracy")
    plt2 = plt.xlabel("Iterations")
    plt2 = plt.title("ETA = " + str(eta))


    for i in range(ITERS):

        old_loss = loss(weights=w, features=X, labels=y)

        ## implement the gradient descent step here
        w += - ETA * grad(w, X, y)

        new_loss = loss(weights=w, features=X, labels=y)
        acc = accuracy(X= X, w= w, y=y)
        plt2 = plt.scatter(i, acc)
```

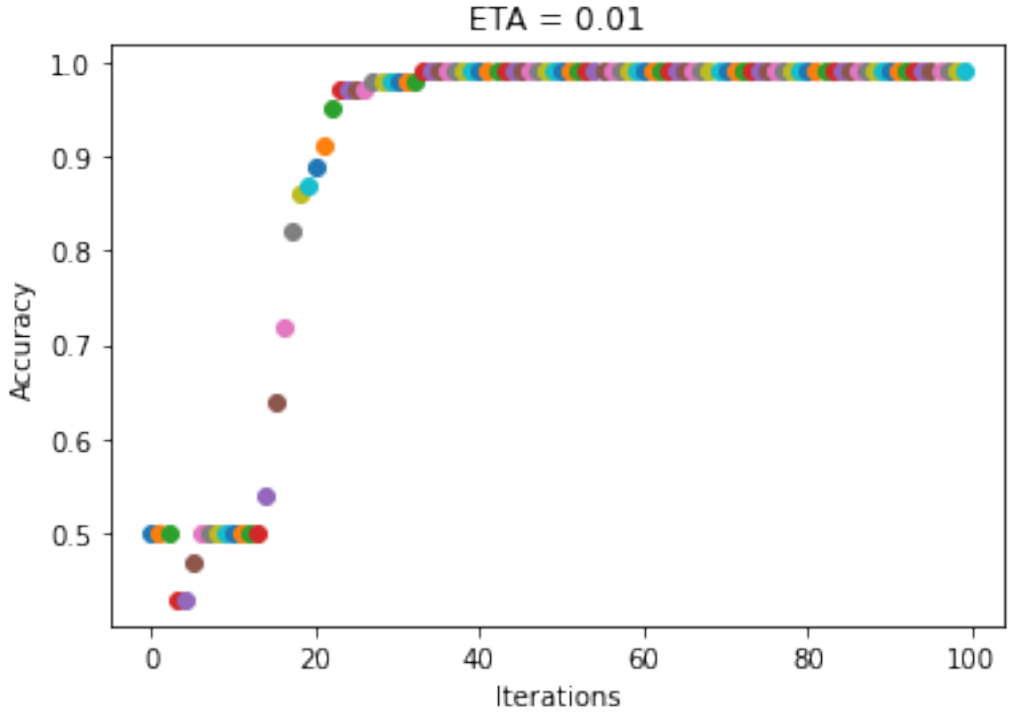
```
        ## the new_loss should be smaller than the old_loss, if you used eta = .
↪0001
GD(0.0001)
```



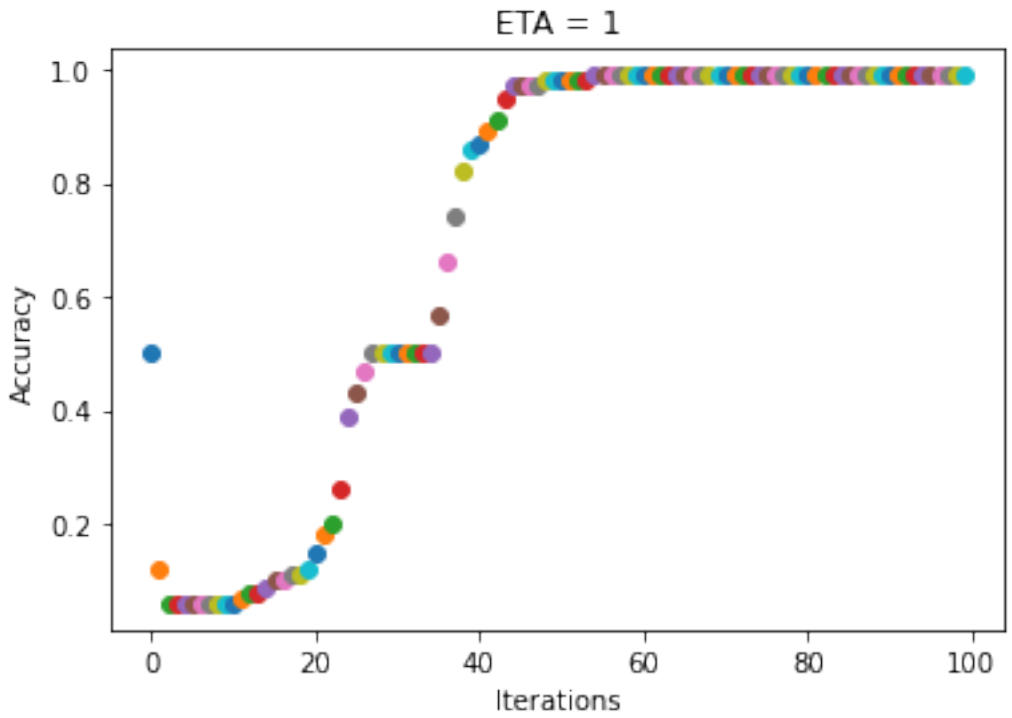
[61]:
```
GD(0.000001)
```


ETA = 1e-06
Accuracy
Iterations
1.0
0.9
0.8
0.7
0.6
0.5
0
20
40
60
80
100

[62]:
```
GD(0.01)
```


ETA = 0.01
Accuracy
Iterations
1.0
0.9
0.8
0.7
0.6
0.5
0
20
40
60
80
100

[54]:
```
GD(1)
```



[ ]:
```

```