CS111: Gradient Descent

QUESTION 1: DATA EXPLORATION

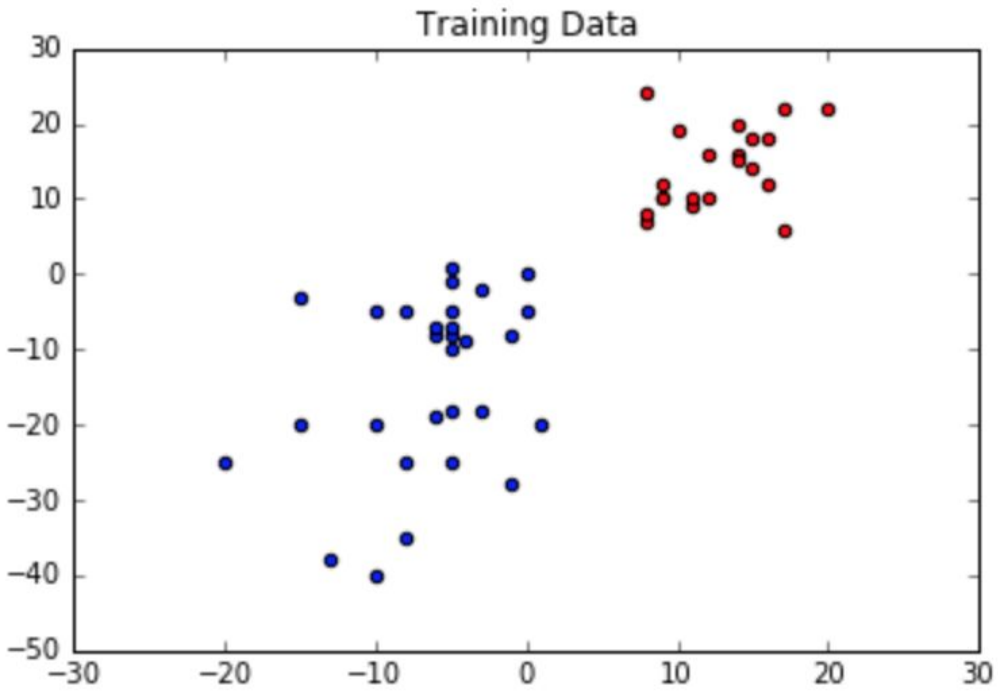
The first step is to get some idea of what the data looks like.

```
import numpy as np
import matplotlib.pyplot as plt
import math
from scipy.spatial import distance
% matplotlib inline

# Import data
df = np.genfromtxt('data.csv', delimiter=',', dtype=int)[1:]

print df[0:5]
```

```
[[ 9 10  0]
 [ 8  7  0]
 [ 8  8  0]
 [11  9  0]
 [12 10  0]]
```

We can see that there are 3 columns for each data point, and X1, and X2, and a Classification
The next step is to graph the data to see it visually.

```
# Define a sorting data function
def sort_data(data, column, val):

    matching_indeces = np.where(data[:, column] == val)
    return matching_indeces, data[matching_indeces]

# Sort the data by classification
neg_indeces, negative_data = sort_data(df, column=2, val=0)
pos_indeces, positive_data = sort_data(df, column=2, val=1)

# Plot the data, red is classified 0, blue is classified 1
plt.scatter(negative_data[:,0],negative_data[:,1], c = "red")
plt.scatter(positive_data[:,0],positive_data[:,1], c = "blue")
plt.title("Training Data")

plt.show
```



QUESTION 2: Generalized Gradient Descent

The next step is to make a generalized gradient descent function. I create it, and test it with a simple function.

```
# A generalized Gradient Descent function for arbitrary dimensions
def grad_descent(fun, part_array, start_array, step_size ):

    """ Gradient Descent
    Takes as inputs a loss function, an array of partial derivatives
    of the loss function, a starting array, and a step size,
    returns the final array of values that minimize the loss function"""

    current_array = start_array
    # set the number of variables (by using length of starting array)
    l = len(start_array)
    counter = 0
    # set previous step size to a number greater than 0.01, so the while loop
will start
    previous_step_size = 1
    # create a loop of the algorithm, stopping at set number of iterations or
while the
    # euclidean distance between two outputs is less than 0.00001
```

```
    while previous_step_size > 0.00001 and counter < 100000:
        prev_array = current_array
        temp_array= []
        for x in range(0,l):
            q = current_array[x]-
step_size*part_array[x](tuple(current_array[0:l]))
            temp_array.append(q)
        current_array = temp_array
        counter +=1
        previous_step_size = distance.euclidean(current_array,prev_array)
    return current_array


# The function to be minimized, formatted as a lambda function with a tuple
input of arbitrary length
f = lambda (x,y,z): (x+5)**2 + (y-2)**2 + z**2
# The corresponding partial derivatives of the above function, formatted the
same way.
p = [lambda (x,y,z): 2*(x+5), lambda (x,y,z): 2*(y-2), lambda (x,y,z): 2*z ]
# An initial starting point for the algorithm, formatted as a list
s = [0,0,0]

#Test the Gradient Descent
current_array = grad_descent(f,p,s,0.01)

print(current_array)
print (f(tuple(current_array[0:3])))
```

[-4.999548985090814, 1.9998195940363255, 0.0]
2.35960760037e-07

QUESTION 3: IMPLEMENT SVM

The first step is to clean our data, as the SVM requires classifications of -1 and 1, not 0 and 1.

```
# Clean: change 0 to -1
negative_data[neg_indeces, 2] = [-1]*len(negative_data)
df[neg_indeces, 2] = [-1]*len(negative_data)
```

We now need to understand what function we are trying to minimize, and what are partial derivatives are. Our SVM is learning the line

$$f(x_1, x_2) = m_1 x_1 + m_2 x_2 + b.$$

```
def f(x1,x2,m1,m2,b):
    return m1*x1 + m2*x2 + b
```

That can separate and classify our data.

In order to test how well it is doing, we define a loss function.

$$L_f(x_i, y_i) = \ln(1 + e^{-y_i f(x_i)}).$$

```
def L(x1,x2,y1,m1,m2,b):
    return math.log(1+math.e**(-y1*((m1*x1) + (m2*x2) + b)), math.e)
```

However, this loss function only finds the loss for one datapoint, and we want to find it for all points in our data frame. Thus we create an error function.

$$E(m_1, m_2, b) = \frac{1}{N} \sum_{i=1}^{N} L_f(x_i, y_i).$$

```
def S(df,m1,m2,b):
    l = len(df)
    li = []
    for i in range(0,l):
        r = L(df[i][0],df[i][1],df[i][2],m1,m2,b)
        li.append(r)
    return np.mean(li)
```

The next step is to find the partial derivatives of each of the functions. We first can calculate the partial derivatives of M1, M2 and B for our loss function, as the sum of the partial derivatives will equal the partial derivative of the error function.

```
def Lm1(x1,x2,y1,m1,m2,b):
    return ((float(-y1)*x1)/(1+math.e**(y1*(m1*x1 + m2*x2 + b))))

def Lm2(x1,x2,y1,m1,m2,b):
    return ((float(-y1)*x2)/(1+math.e**(y1*(m1*x1 + m2*x2 + b))))

def Lb(x1,x2,y1,m1,m2,b):
    return (float(-y1)/(1+math.e**(y1*(m1*x1 + m2*x2 + b))))
```

```
def Sm1(df,m1,m2,b):
    l = len(df)
    li = []
    for i in range(0,l):
        r = Lm1(df[i][0],df[i][1],df[i][2],m1,m2,b)
        li.append(r)
    return np.mean(li)

def Sm2(df,m1,m2,b):
    l = len(df)
    li = []
    for i in range(0,l):
        r = Lm2(df[i][0],df[i][1],df[i][2],m1,m2,b)
        li.append(r)
    return np.mean(li)

def Sb(df,m1,m2,b):
    l = len(df)
    li = []
    for i in range(0,l):
        r = Lb(df[i][0],df[i][1],df[i][2],m1,m2,b)
        li.append(r)
    return np.mean(li)

p = Sm1, Sm2, Sb
```

The next step is to modify our generalized gradient descent. We modify it so that it can accept a data frame for the error function to work, and change the stopping condition to be change in our error function. We also include a line to print the current state of the SVM at iterations 1, 10, 100, and multiples of 1000.

```
def grad_descent(df, fun, part_array, start_array, step_size ):
    current_array = start_array
    l = len(start_array)
    counter = 0
    # set Delta Error Function to a number greater than 0.01, so the while loop
will start
    d_e_f = 1
    while d_e_f > 0.000001 and counter < 50000:
        prev_array = current_array
        temp_array= []
        for x in range(0,l):
            q = current_array[x]-
step_size*part_array[x](df,current_array[0],current_array[1],current_array[2])
            temp_array.append(q)
        current_array = temp_array
```

```
        counter +=1
      # Print output of the descent at certain iterations
       if counter % 100 == 0 or counter ==1 or counter == 10:
            print current_array,
                  fun(df,current_array[0],current_array[1],current_array[2])
      #Set the stopping condition to the change in error function
        d_e_f = abs(fun(df,prev_array[0],prev_array[1],prev_array[2]) -
                    fun(df,current_array[0],current_array[1],current_array[2]))

    return current_array,
           fun(df,current_array[0],current_array[1],current_array[2])
```

We can now run our function.
We start with (0,0,0) as our starting array, and set our step size to 0.01

```
sp = [0,0,0]
current_array, error= grad_descent(df,S,p,sp,0.01)
print current_array
print error
```

```
[-0.045100000000000001, -0.07110000000000001, 0.00080000000000000004] 0.260636616361
[-0.11898899776848315, -0.1598415249049831 6, 0.0064679099564155639] 0.0989506318064
[-0.26943740287010465, -0.3016986332589035, 0.041042720479046016] 0.0375821355548
[-0.33839296243576639, -0.3554136052644837 3, 0.067572713669629372] 0.0290753748
[-0.38544200662154715, -0.3893609730729099, 0.089738732252871378] 0.0251950754609
[-0.42193681538056393, -0.4146621704306199 3, 0.10945007498297234] 0.0228281569756
[-0.45197471432830533, -0.4350217889603600 3, 0.12752001466543184] 0.0211824597487
[-0.47757895619147789, -0.4521516699319493 3, 0.14438941871898756] 0.019947816577
...
[-0.77247967935388062, -0.65156184511887749, 0.51343169716257608] 0.0108112225751
[-0.77651568971793217, -0.6544316133894008 1, 0.52229534510105746] 0.0107081381245
[-0.77923735209186074, -0.6563697009810808 3, 0.52836650475486124]
0.0106385439675
```

Our function has produced a result, with a low overall error! However, it is hard to gauge intuitively how well it worked without a graphical representation.

The function

$$f(x_1, x_2) = m_1 x_1 + m_2 x_2 + b$$

Initially appears to be hard to graph. However, what we are really looking to graph is the line

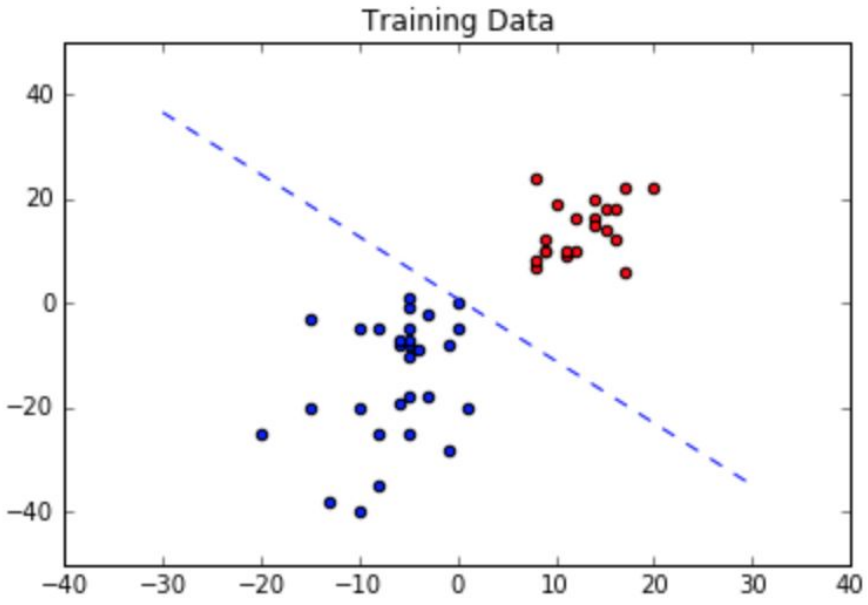
$$0 = m_1 x_1 + m_2 x_2 + b$$

Which can be rearranged as

$$x_2 = \frac{-m_1 x_1 - b}{m_2}$$

Which can be graphed.

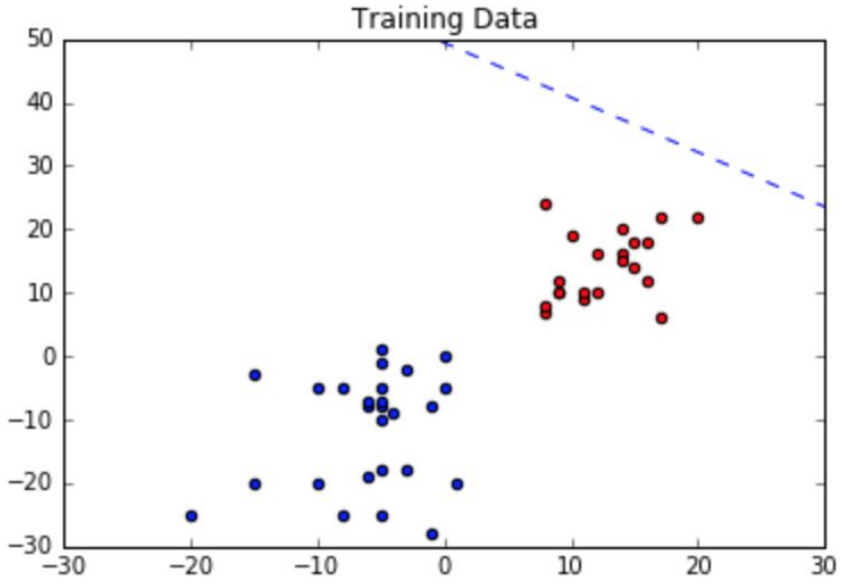
```
# Define a function to graph the line
def line((m1,m2,b)):
    axes = plt.gca()
    x_vals = np.array(axes.get_xlim())
    y_vals =  (-m1*x_vals-b)/m2
    plt.plot(x_vals, y_vals, '--')


plt.scatter(negative_data[:,0],negative_data[:,1], c = "red")
plt.scatter(positive_data[:,0],positive_data[:,1], c = "blue")
plt.title("Training Data")
line (tuple(current_array[0:3]))
plt.show()
```



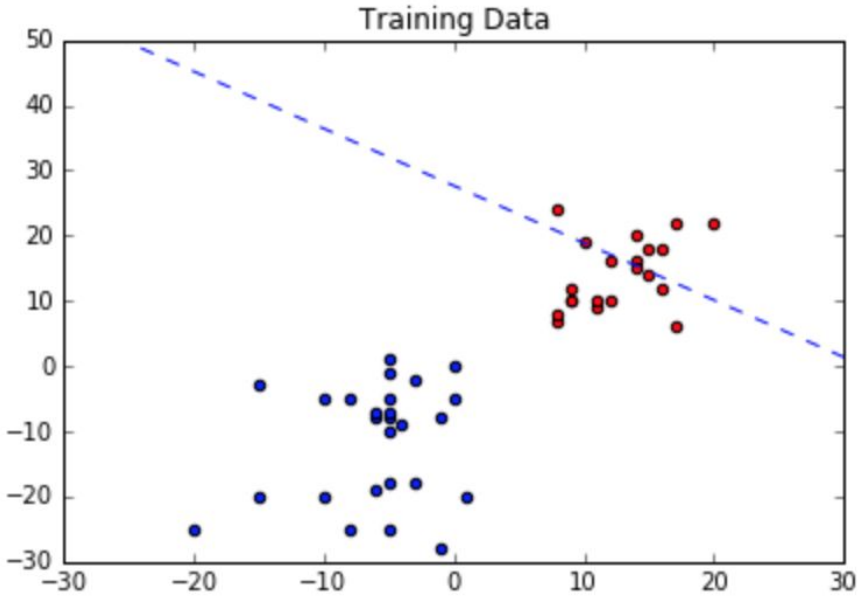
EXTENSION

The above line clearly works as a separating line for the data. However, we don't know what line we started with, and the line is quite close to one of the training values. Due to the nature of gradient descent, multiple lines can be found due to different local minima being found. By varying the starting conditions, we can get different outputs for our line. We can also graph the function after different time steps. The best result I got was by starting with initial conditions (0,0,3).

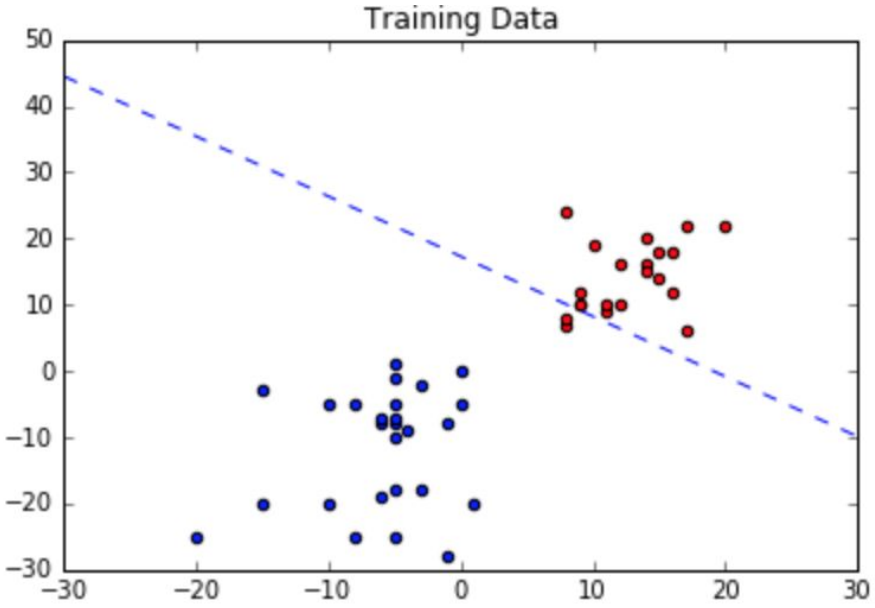
ITERATIONS = 1



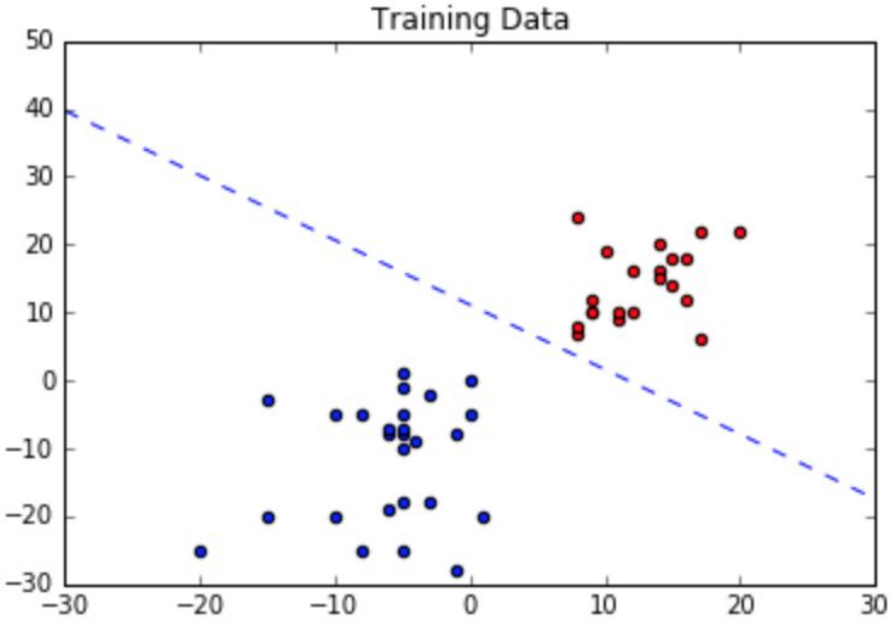
ITERATIONS = 2



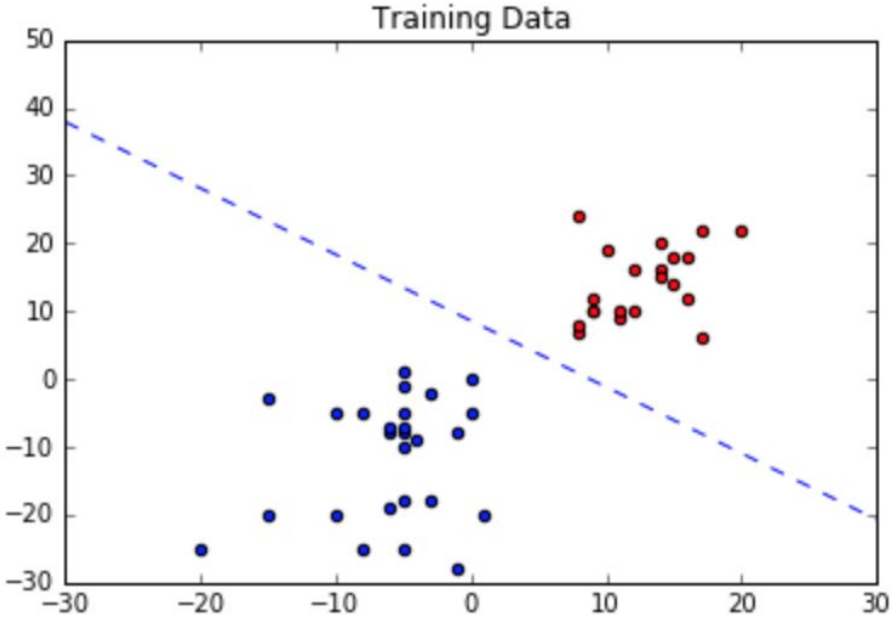
ITERATIONS = 5



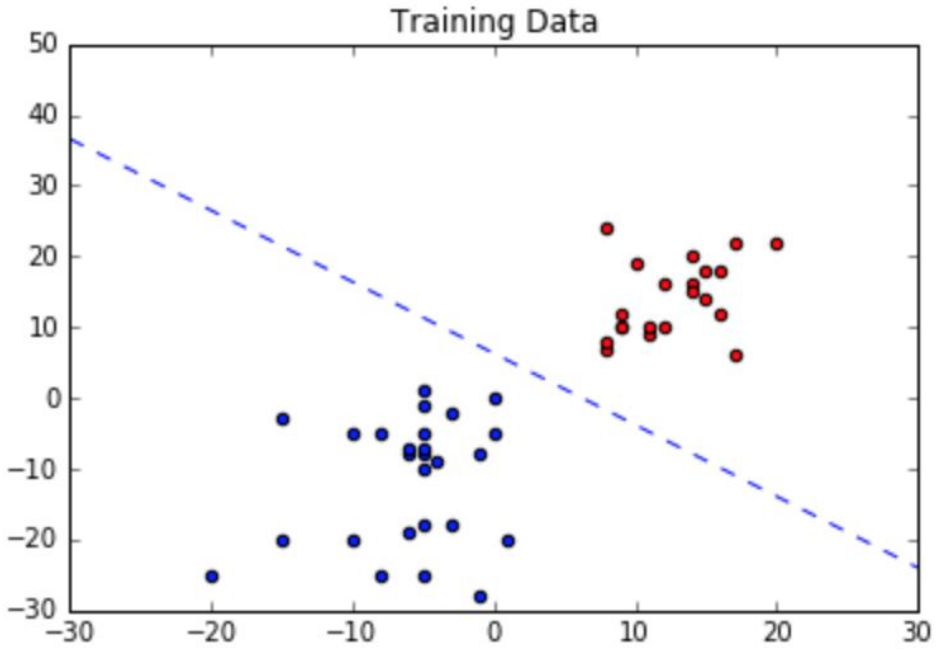
ITERATIONS = 25



ITERATIONS = 100



ITERATIONS = 893 (FINAL)



QUESTION 4: LOSS FUNCTION

Question 4 concerns the loss function of

$$L_f(x_i, y_i) = \ln(1 + e^{-y_i f(x_i)}).$$

In order to have a both a SVM and a gradient descent, we need to have a loss function. The gradient descent functions by finding a minimum of this function. In the case of the SVM, the goal is classification. We need to classify any new points into one of two categories, in this case -1 and 1. Each proposed set of *M1, M2,* and *B* give us a different line that is classifying any new points. If it is above the line, it is classified as 1, and if it is below the line, it is classified as -1. The goal is to have a line that will classify all points as correctly as possible. We start with an arbitrary line, and use gradient descent to get to a line that does this. Thus, we need an equation that tells us how wrong or right we are with a given point and line. This is our loss function.

The log loss function is a clever way to do this.
The *e* term is raised the the power $-y_i f(x_i)$

While *y* represents the true classification, *f(x)* represents our guess. If *f(x)* is positive, we guess that the classification is 1 and when *f(x)* is negative, we guess that the classification is negative.

Thus, when our guess is correct (it's really 1, we guess 1), *e* will be raised to a negative exponent, and as *f(x)* increases, the term

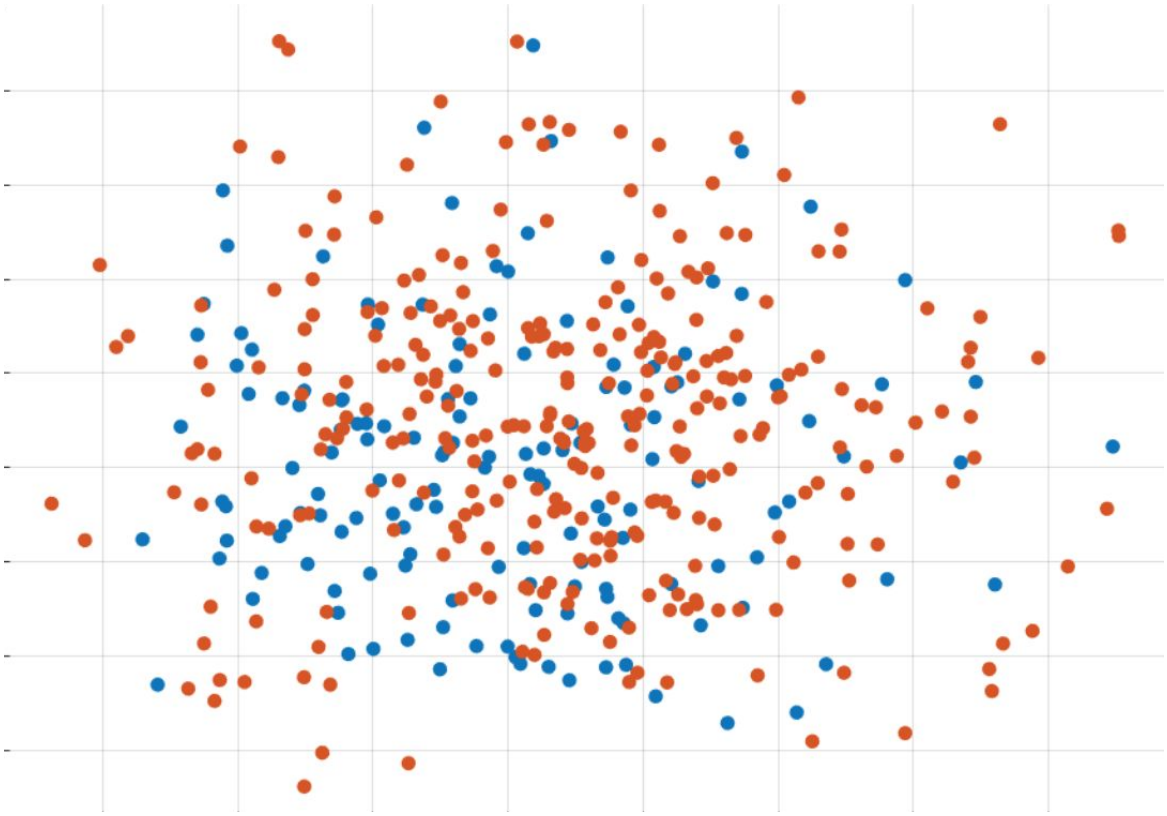
$$(1 + e^{-y_i f(x_i)})$$

Approaches 1. The natural log of 1 is 0, and so our loss function will near zero. Simply, the more right we are, the closer our loss function will be to zero (with rightness being defined as distance away from the line - if our point is close to the line, we were close to misclassifying it, but if it is far above the line, we were not close to misclassifying and thus were quite correct.

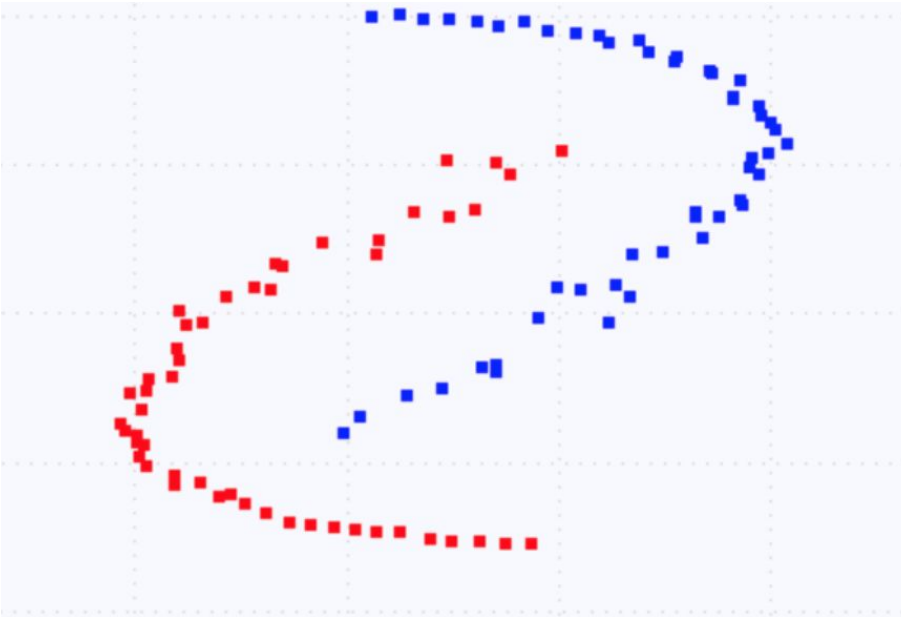
The opposite is also true. If the result of *f(x)* has an opposite sign as *y*, the *e* term will be raised to a positive exponent, and thus as *f(x)* increases the term we are taking the natural log of increases. Thus, our loss function tells us when we have misclassified something, and increases as we have misclassified by a larger and larger number.

QUESTION 5: DATA THAT DOESN'T WORK

This SVM works well extremely well when we have data that can be classified into two groups by a hyperplane (in our case a line). Thus, it is possible to have a dataset that will not give us a good classifying line. If we have a dataset that is randomly arranged with no clear dividing line, our program won't work. For example, this data set



Another problem would be data that has a clear distinction, but that can't be divided by one straight hyperplane. This is unfortunate as the data could be classified, but is not because of limits with our algorithm. An example of this kind of data set is



The data is clearly divided, but no straight line could split it by color.

QUESTION 6: SPAM FILTER

The basic idea is quite similar with a SPAM filter. The filter is meant to classify an email into one of two categories, SPAM or not SPAM. Certain aspects of the email (which I will discuss below) will be used as aspects of each data point, and then a hyperplane would be created with an SVM that decides if it is SPAM or not. The best way to do this would be to get a large data set of SPAM emails and normal emails, to learn what actually makes a SPAM email. Aspects such as word length, word diversity, number of emails from this address, number of attachments, and number of a list of keywords would be used to create our SVM, just as with *X1* and *X2* in our previous example. We would then create a multi dimensional graph of all of the variables, and create a hyperplane that divides them into two groups, SPAM or not. The loss function would need to tell us (when training the model) if we classified it right and how far off we were. Then we would run a gradient descent, find an optimal hyperplane, and use this to classify our emails in the future.