# CS111 Assignment

## Nonlinear Regression via Gradient Descent

### Prof. Drummond

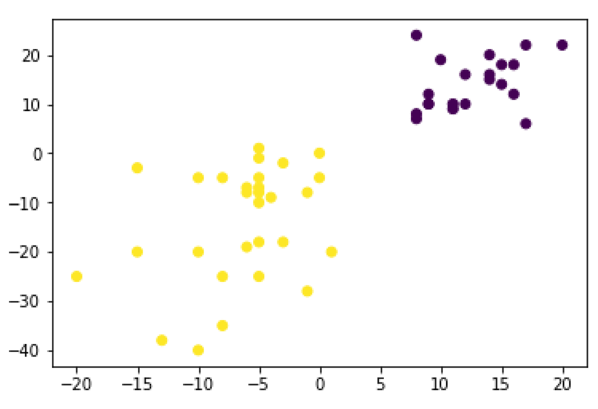
Tuan Tran - Spring 2018

### 1. Data exploration

In [3]:
```python
import csv, math
import numpy as np
import matplotlib.pyplot as plt
```

In [5]:
```python
with open('cs111-svm-dataset---sheet1.csv') as csvfile:
    data = list(csv.reader(csvfile, delimiter=','))

data = np.array(data[1:], dtype = np.float)
```

In [12]:
```python
x1 = data[:,0]
x2 = data[:,1]
y = data[:,2]
plt.scatter(x1,x2, c= y)
plt.show()
```



Our original dataset comprises of 50 observations. We have 2 independent variables (X1 and X2) and an independent variable, which classifies our the data into 2 classes (0 and 1). The graph illustrates that the data are groups into 2 clusters, purple (representing data points with value 0) and yellow (representing data points with value 1).

### 2. Gradient descent

The code below runs the gradient descent for a function of 3 variables (m1, m2, b) to generate an SVM which is the equation of the separating line. We are trying to minimize the value of the error function, which is the average loss across all observations. Instead of using a for loop to examine every data point, I use operations on matrices using the numpy library to implement it effciently.

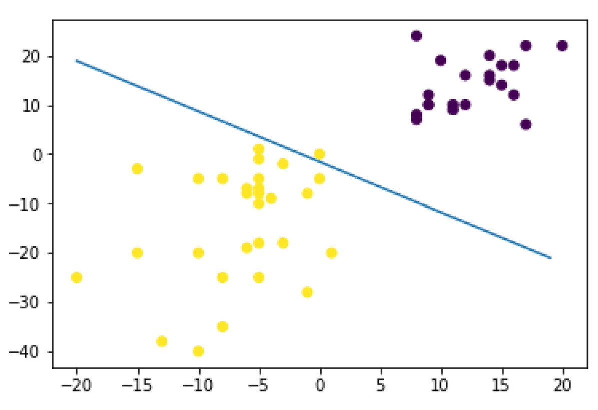
In [69]:
```python
def error_function(m1, m2, b,f):
    """
    Return the value of the error function
    """
    error = np.mean(np.log(1  + np.exp(-y*f)))
    return error

def gradient(m1,m2,b,f):

    """
    Return a list of gradient corresponding to current
    value of m1, m2, b
    """

    gradient_m1 = np.mean((1/(1 + np.exp(-y*f))) * np.exp(-y*f) * (-y)*x1)
    gradient_m2 = np.mean((1/(1 + np.exp(-y*f))) * np.exp(-y*f) * (-y)*x2)
    gradient_b = np.mean((1/(1 + np.exp(-y*f))) * np.exp(-y*f) * (-y))

    lis = [gradient_m1, gradient_m2, gradient_b]
    return lis

def plotting(m1,m2,b):
    """
    Plotting the data and the current SVM at each step
    """
    global x1, x2, y
    plt.figure()
    plt.scatter(x1,x2, c= y)
    abline_values = [-m1/m2 * x1 - b/m2 for x1 in range(-20,20)]
    plt.plot(range(-20,20), abline_values)
    plt.show()

def gradient_descent(guess, step_size, max_steps):
    global x1, x2, y
    #main program
    m1 = guess[0]
    m2 = guess[1]
    b =  guess[2]

    errors = []
    for step in range(max_steps):
        #a vector contains predicted y values for all x1 and x2 based on current m1,m2,b
        f= x1*m1 + x2*m2 +b

        gradient_list = gradient(m1,m2,b,f)
        temp_m1 = m1 - step_size*gradient_list[0]
        temp_m2 = m2 - step_size*gradient_list[1]
        temp_b = b - step_size*gradient_list[2]

        m1, m2, b = temp_m1, temp_m2, temp_b #simultaneous update
        errors.append(error_function(m1, m2, b,f))

        if step% 300 ==0 or step ==999: #plot the graph of the SVM each 300 steps
            print("Current step: {}".format(step))
            print("Current error: {}".format(error_function(m1, m2, b,f)))
            print("m1 = {:f}, m2 = {:f}, b = {:f}".format(m1,m2,b))
            plotting(m1,m2,b)
    return errors
```

### 3. Implementation

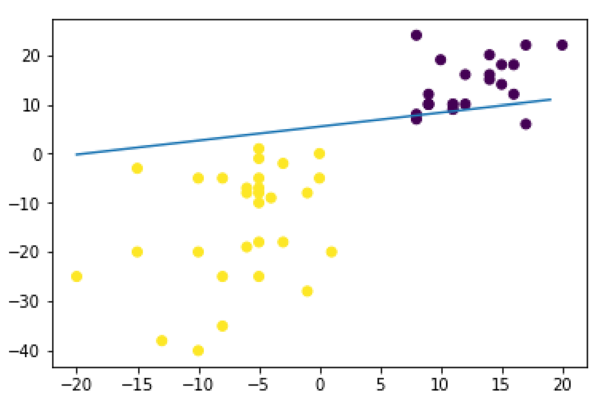
I implement the gradient descent for 1000 steps and a step size of 0.01 to find the SVM for our data set. I plot the line f learned at each 300 steps to visually demonstrate the algorithm. I also plot the error function over time to observe how the algorithm minimize the error values.

In [70]:
```python
initial_guess = [2,2,3]
errors = gradient_descent(initial_guess, 0.01, 1000)
```

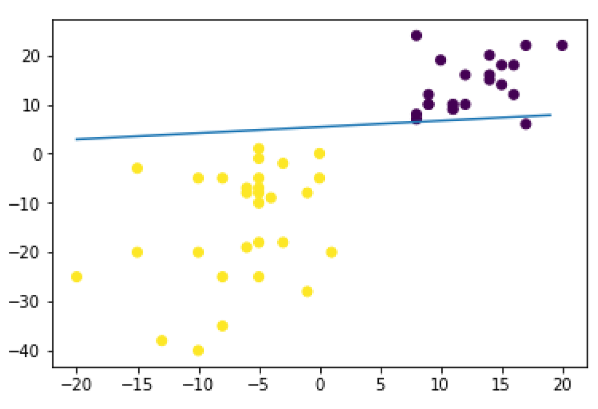
```
Current step: 0
Current error: 22.572266803508132
m1 = 1.962807, m2 = 1.917400, b = 3.005608
```



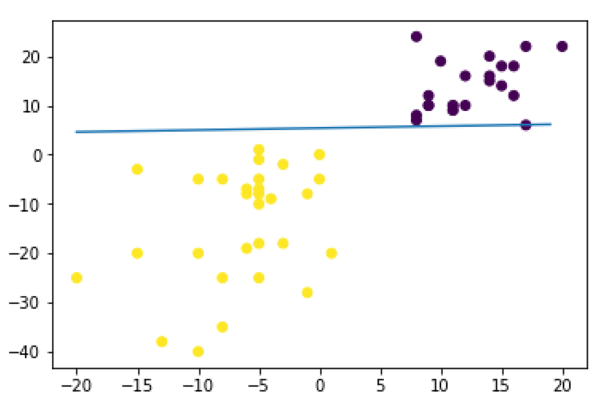
```
Current step: 300
Current error: 0.2985686528003694
m1 = 0.167230, m2 = -0.588498, b = 3.232990
```



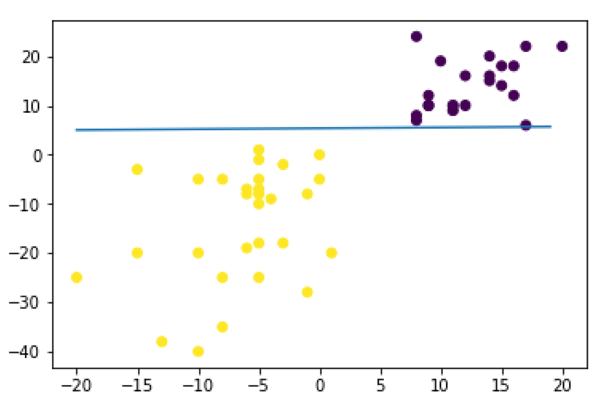
```
Current step: 600
Current error: 0.295471264760855
m1 = 0.075748, m2 = -0.600102, b = 3.248865
```



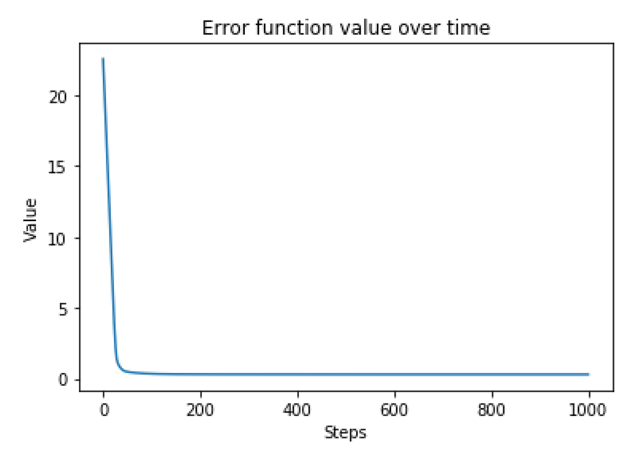
```
Current step: 900
Current error: 0.29451616343733794
m1 = 0.024023, m2 = -0.605108, b = 3.260000
```



```
Current step: 999
Current error: 0.29431896117018413
m1 = 0.010484, m2 = -0.606273, b = 3.263185
```



In [52]:
```python
plt.plot([step for step in range(1000)], errors)
plt.title("Error function value over time")
plt.xlabel("Steps")
plt.ylabel("Value")
plt.show()
```




As we can see, the error value decreases sharply and reaches an asymptotic value of approximately 0.30 after about 20 steps. In general, the time it takes for the error value to converge depends on several factors:

- initial guess: choosing an unsuitable initial guess can make the function take longer time to converge. We also run the risks of getting stuck in local optimum. Therefore, we can examine the data by plotting it initially to choose a sensible initial value for m1, m2 and b.
- step size: A small step size takes longer time to converge. However, an exceedingly big step size also causes the algorithm to overshoot and even diverge. We can experiment with different step sizes to find the most optimal one.

### 4. Interpretation of the Loss Function

- The lost function is an indication of the accurarcy of the SVM (our classification algorithm). Suppose that an observation $x_o$ have an $y$ value of 0, the SVM is accurate if it classifies $x_o$ as having the $y$ value of 0 and inccorect otherwise.
- To be more specific, the loss function will return a small value if the classification is correct since $e^{-y \cdot f(x)}$ would be small in this case. On the other hand, it returns a large value if the classification is incorrect since $e^{-y \cdot f(x)}$ would be large in this case.

Due to this property, we can implement gradient descent to minimize error function, which is the average of the loss function across all observations, to find the SVM that can classify the data best.

### 5. SVM strengths and weaknesses

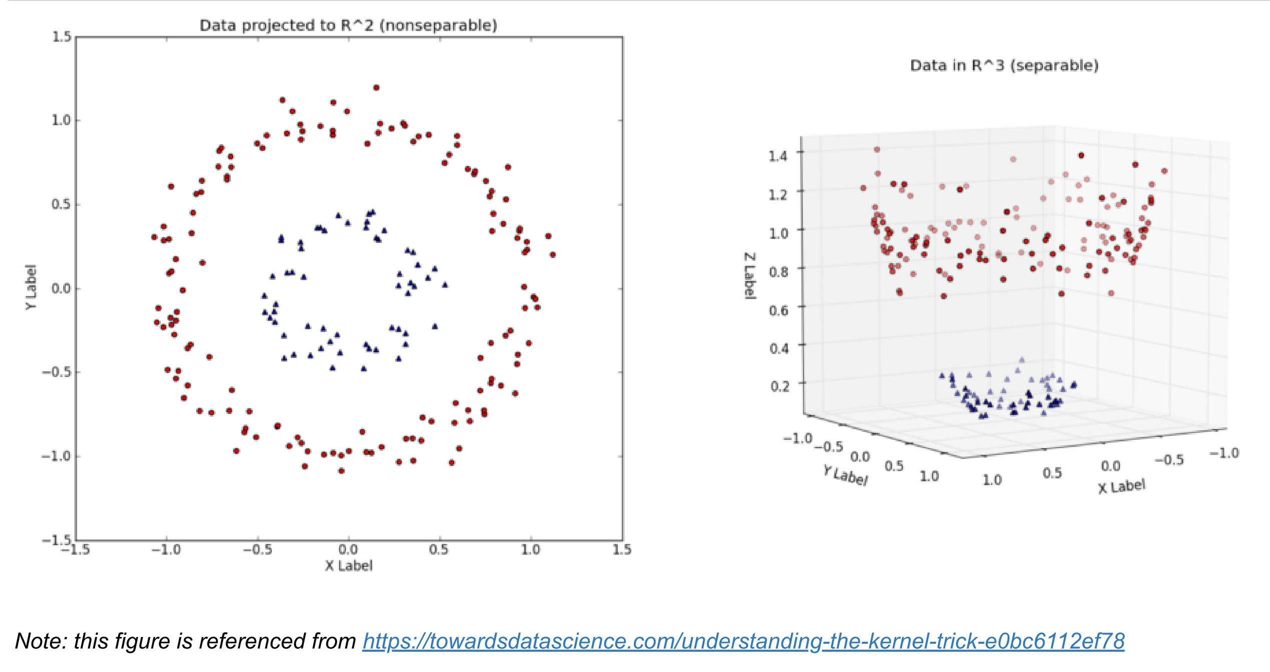
The strength of our algorithm and SVM is that it is quite easy to implement and we do not need to calculate an analytical solution, which can be hard to do when we have millions of data points. Using gradient descent, we can also specify the step size at our disposal.

However, a major disadvantage of SVM is that it can only classify **linearly separable data**. For example in this case, a line is sufficient to classify the data. On the other hand, for data that cannot be separated linearly, SVM classification can be quite inaccurate.

In this case, we can implement **kernel trick** to transform our existing data into a higher dimensional data, which can help us classify the data better. For example in the figure below, the data is not linearly separable (left) but we can map them into a 3-dimensional space (right) where a separating hyperplane can be found.

In [71]:
```python
from IPython.display import Image
Image("Kerneltrick.png")
```

Out[71]:




*Note: this figure is referenced from https://towardsdatascience.com/understanding-the-kernel-trick-e0bc6112ef78*

### 6. SVM for Spams Classification

We can specify some variables that are typical for an email to classify whether it is a spam or not. For example:

- x1 - Frequency of words: Spams are usually used in promotion. Therefore, they are more likely to contains clickbait words such as "Discounts", "Free Trials", "Click here". We can build a list of all these words and calculate the frequency of that those words appear in the emil as a variable in the classifier.
- x2 - Sender: spams email usually comes from senders who are not in the contact list or have mutual connections with a person. It also comes from dubious companies that usually send out promos. However, we should make the weight on this less important to avoid mistakenly classify important email from new people.
- x3 - Number of embeded links: A spam email usually contains many links that users can click on. Usually it's dangerous since it may contain viruses or unwanted content.