Department of Data Science

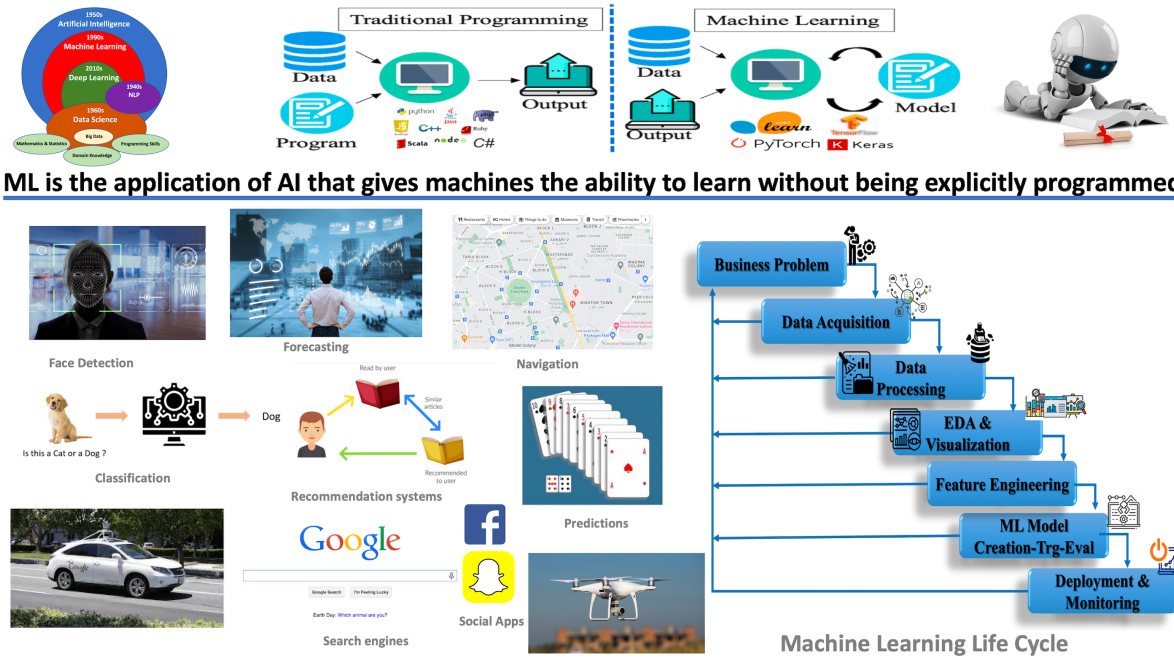
Course: Tools and Techniques for Data Science

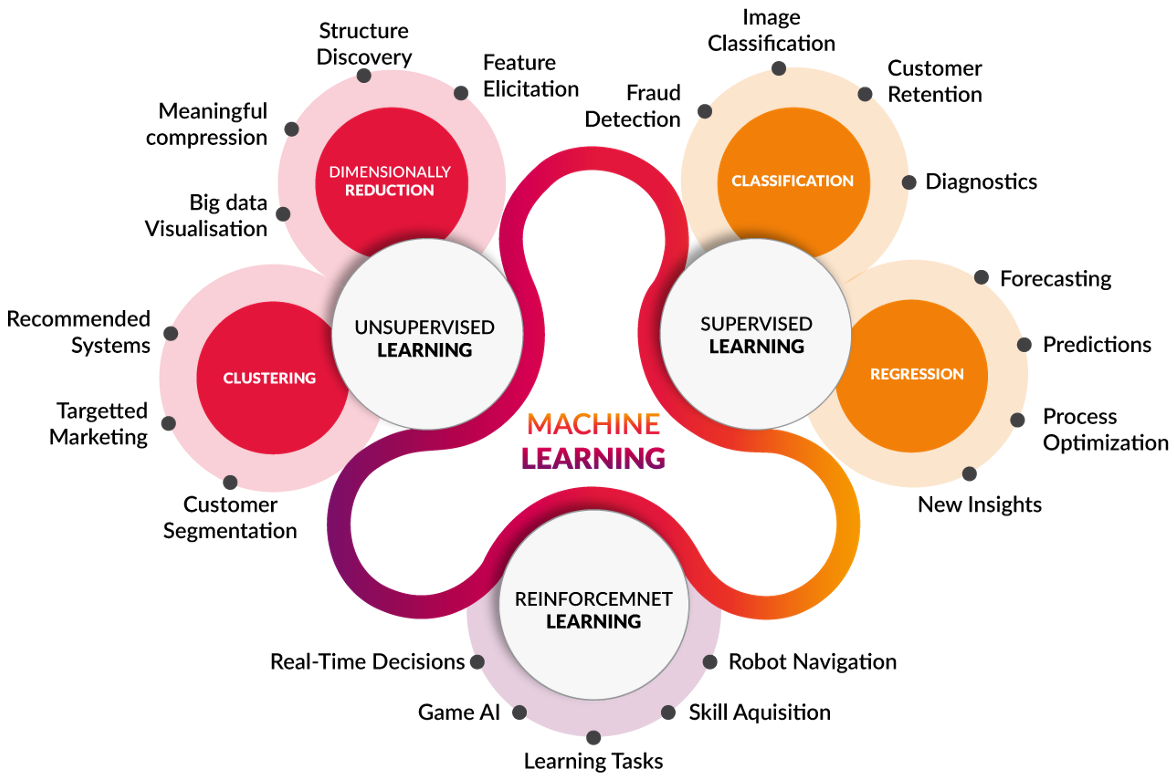
Instructor: Muhammad Arif Butt, Ph.D.

Lecture 6.20 (Logistic Regression Part-I)

Open in Colab

(https://colab.research.google.com/github/arifpucit/data-science/blob/master/Section-4-Mathematics-for-Data-Science/Lec-4.1(Descriptive-Statistics).ipynb)





Learning agenda of this notebook

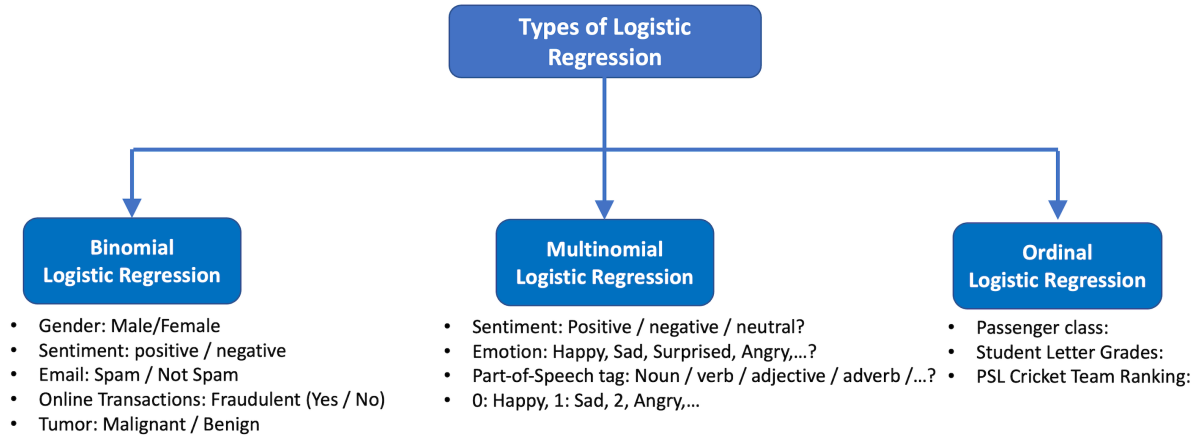
- Logistic Regression (What, Why and How?)
  - Abstract Overview
  - Types of Logistic Regression
  - Linearly Separable Data
  - Assumptions for Logistic Regression
  - From Linear Regression to Logistic Regression
- Understanding Single Layer Perceptron
  - What is a Single Layer Perceptron?
  - How to find the Weights of the Classification Line (Decision Boundry)?
- Logistic Regression using Single Layer Perceptron (`Step Function`)
  - **Sample Code:**
- Logistic Regression using Scikit-Learn Model
  - **Sample Code:**
- Improving Single Layer Perceptron (`Step Function`) Algorithm
  - Limitation
  - How to Overcome this Limitation?
  - What is Sigmoid Function?
- Logistic Regression using Single Layer Perceptron (`Sigmoid Function`)
  - **Sample Code:**

1. Logistic Regression (What, Why and How?)

a. Abstract Overview

Logistic regression is a statistical model that allows us to predict a categorical label based on historical feature data by predicting the probability of an outcome, event, or observation.

b. Types of Logistic Regression

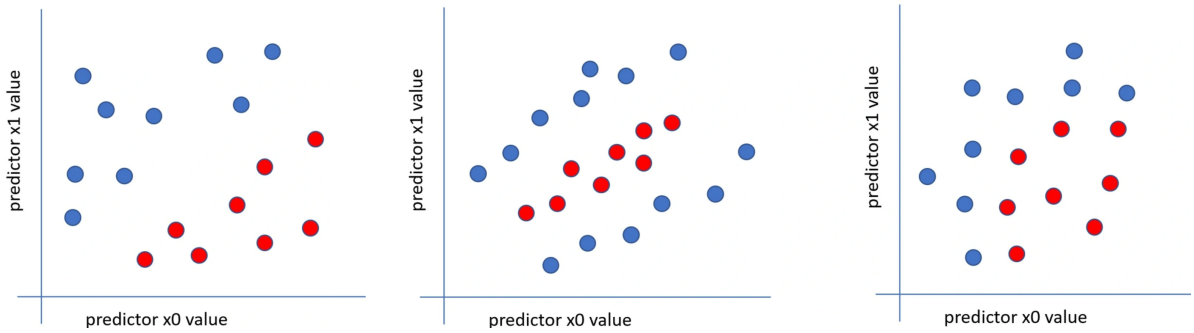


c. Logistic Regression is Linear Model

- In **Regression**, a `Linear Model` means if you plot all the input features and the output variable, then there exist a straight line or hyperplane that rougly estimate the outcome. For single input feature there exist a line, for two input features there exist a plane, and for more than two input features there exist a hyperplane that estimates the outcome. For example $y = 2x + 5$ is a linear model, because the output changes by the same amount for any given change in input (slope is constant). On the contrary $y = x^2 + 3$ is a `Non-Linear Model` , because the output donot change with the same amount for any given change in input (slope is not constant)
- In **Classification**, a `Linear Model` means if you plot all the `n` input features, then there exist a `n-1` dimensional line/hyperplane that can separates different classes. For single input feature there exist a point, for two input features there exist a line, for three input

feature there exist a plane, and for more than three input features there exist a hyperplane that separates the datapoints of different classes.

> **Linearly separable data is data that if graphed in two dimensions, can be separated by a straight line.**



d. Assumptions for Logistic Regression

Four assumptions for **Linear Regression** are `Linearity` (require a linear relationship between the input features and output variable), `Independence` (Input features are independent of eachother), `Homoscedasticity` (variance of the residuals is constants) and `Normality` (residuals are normally distributed). On the contrary **Logistic Regression** does not require `Linearity`, `Homoscedasticity` and `Normality`, however requires `Independence`.

Six assumptions for **Logistic Regression** are:

a. Input features should not be too highly correlated with each other (little or no multicollinearity)

b. Response variable is binary

c. Observation are independent of each other, i.e., the observations should not come from repeated measurements or matched data

d. Prefers large sample size

e. No extreme outliers

f. Feature variables are linearly related to log odds

> - Log odds play an important role in logistic regression as it converts the LR model from probability based to a likelihood based model. More on this later...

e. From Linear Regression to Logistic Regression

A comparison

Similarities:

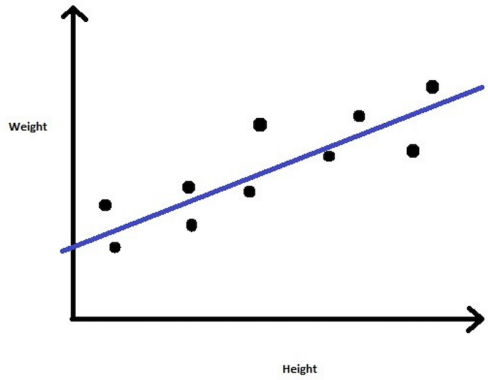
- Linear Regression and Logistic Regression both are `supervised` Machine Learning algorithms.
- Linear Regression and Logistic Regression, both the models are `parametric` i.e. both the models use linear equations for predictions. In general, every Linear Model consists of sum of parameters (regression coefficients or weights) multiplied by independent variables.

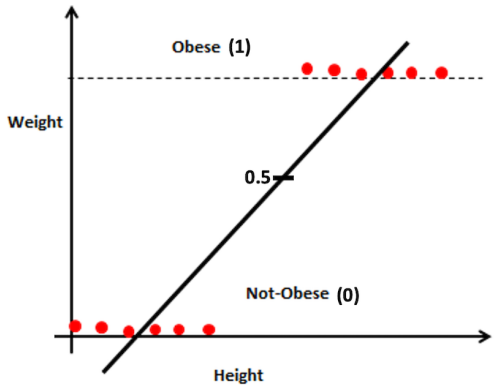
$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_m x_m$$
$$y = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_m x_m$$

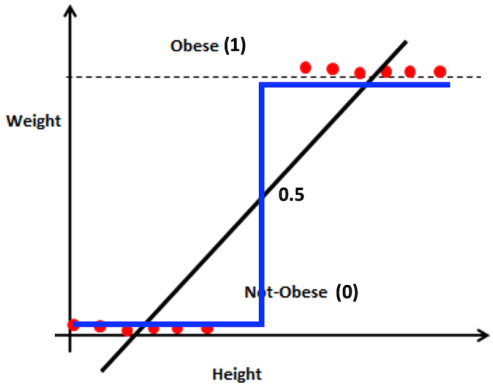
Differences:

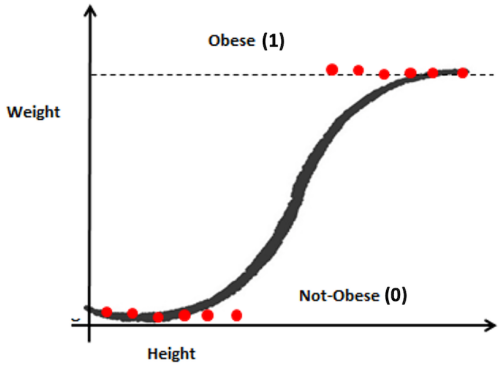
- Linear Regression is used to handle `regression problems` whereas Logistic regression is used to handle the `classification problems`.
- Linear regression provides a `continuous output` but Logistic regression provides `discrete output`.
- The purpose of Linear Regression is to find the `best-fitted line` while Logistic regression is one step ahead and fitting the line values to the `sigmoid curve`.
- The method for calculating loss function in linear regression is the `mean squared error` whereas for logistic regression it is `maximum likelihood estimation`.

Can we use Regression for Classification







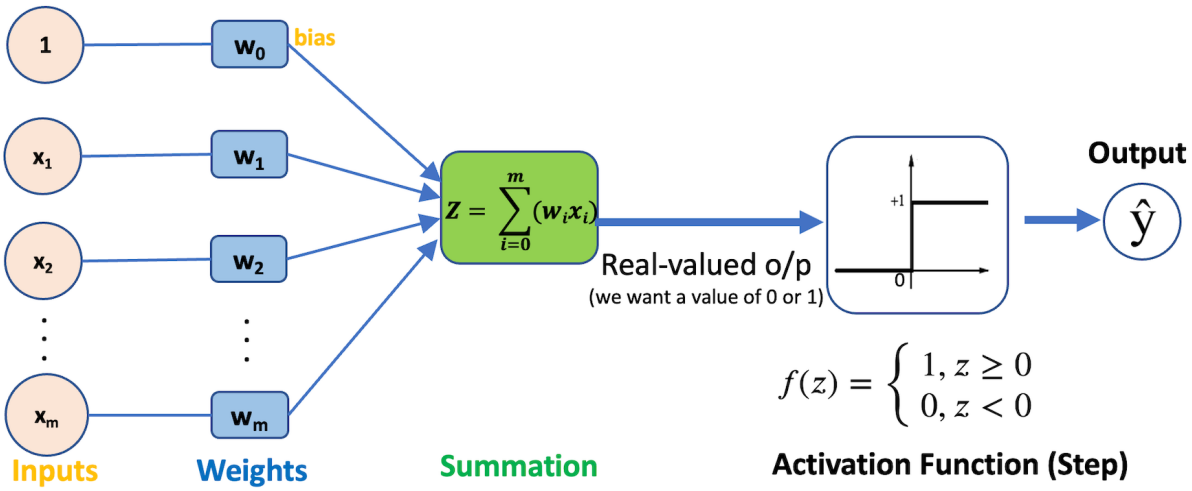


- The straight line in top right graph is susceptible to outliers, while the `step function` (that maps a real value to a discrete value of 0 or 1) shown in bottom left graph handles this limitation.
- The limitation of step function is that it cannot be used for multi-class classification, and the solution is `sigmoid function` (that maps a real value between 0 to ∞ to a real value between 0 and 1). More on this in next lecture...

2. Logistic Regression using Single Layer Perceptron

a. What is a Single Layer Perceptron?

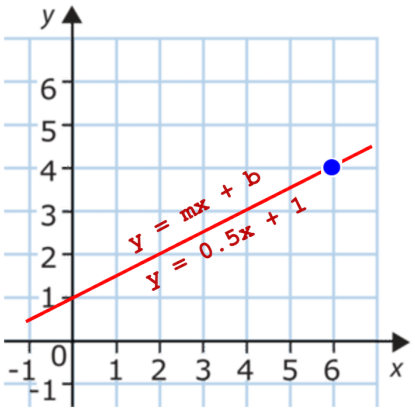
A perceptron is a basic computational unit of a neural network that is designed to perform binary classification of input vectors.



b. How to find the Weights of the Classification Line

(Decision Boundry)?

Two Forms of Equation of a Line



$y = 0.5x + 1$

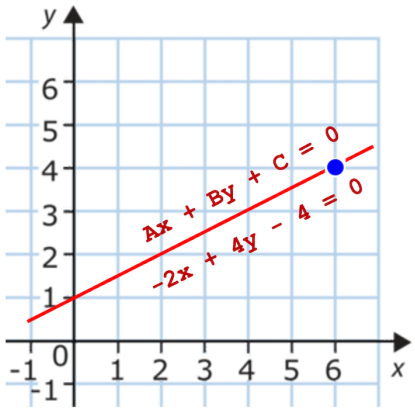
$-0.5x + y - 1 = 0$

$-2x + 4y - 4 = 0$

$A=-2,\ B=4,\ C=-4$

$y = -\frac{A}{B}x - \frac{C}{B}$

$y = -\frac{-2}{4}x - \frac{-4}{4}$



$$Ax + By + C = 0$$

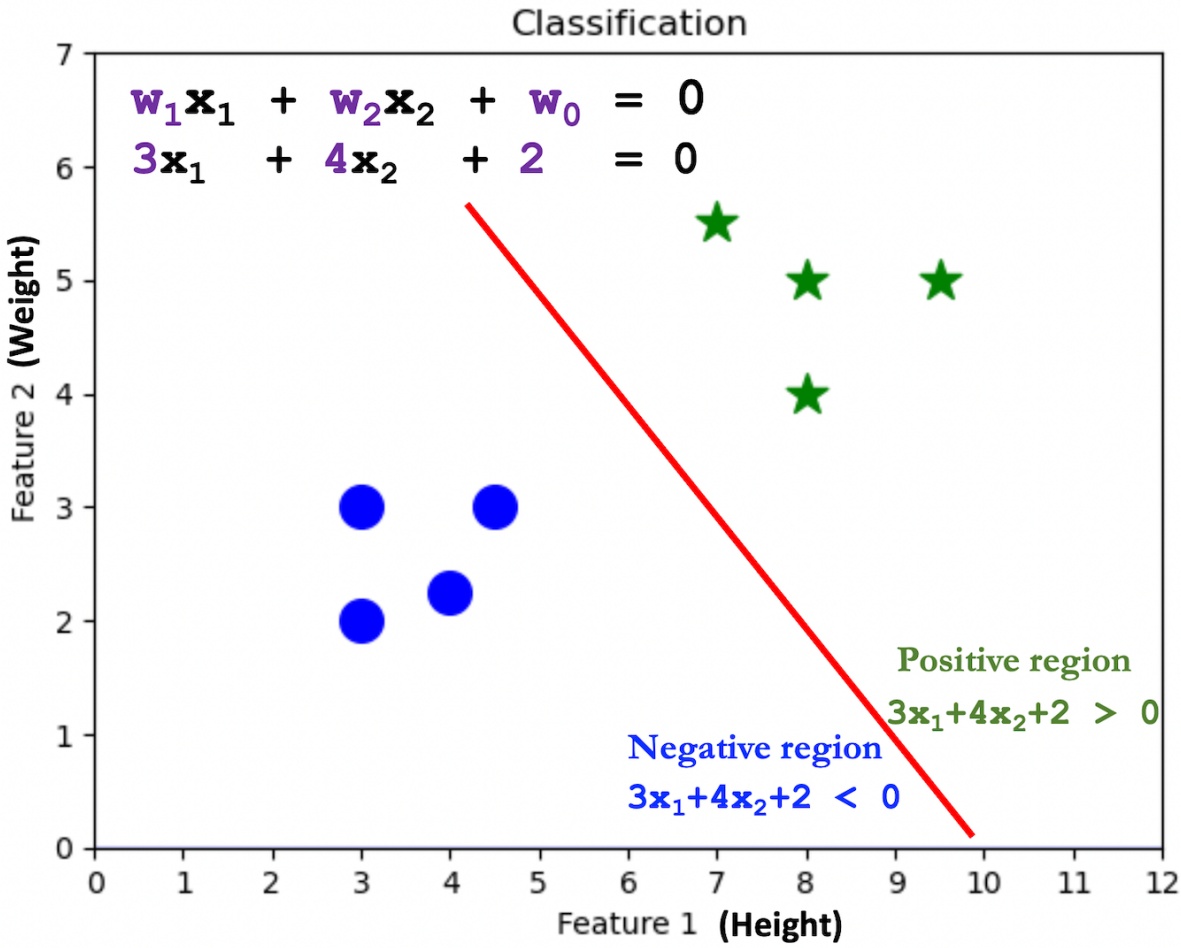
$$Ax_1 + Bx_2 + C = 0$$

$$w_1 x_1 + w_2 x_2 + w_0 = 0$$

$$w_1 x_1 + w_2 x_2 + w_3 x_3 + w_0 = 0$$

$$w_1 x_1 + w_2 x_2 + w_3 x_3 + \cdots + w_m x_m + w_0 = 0$$

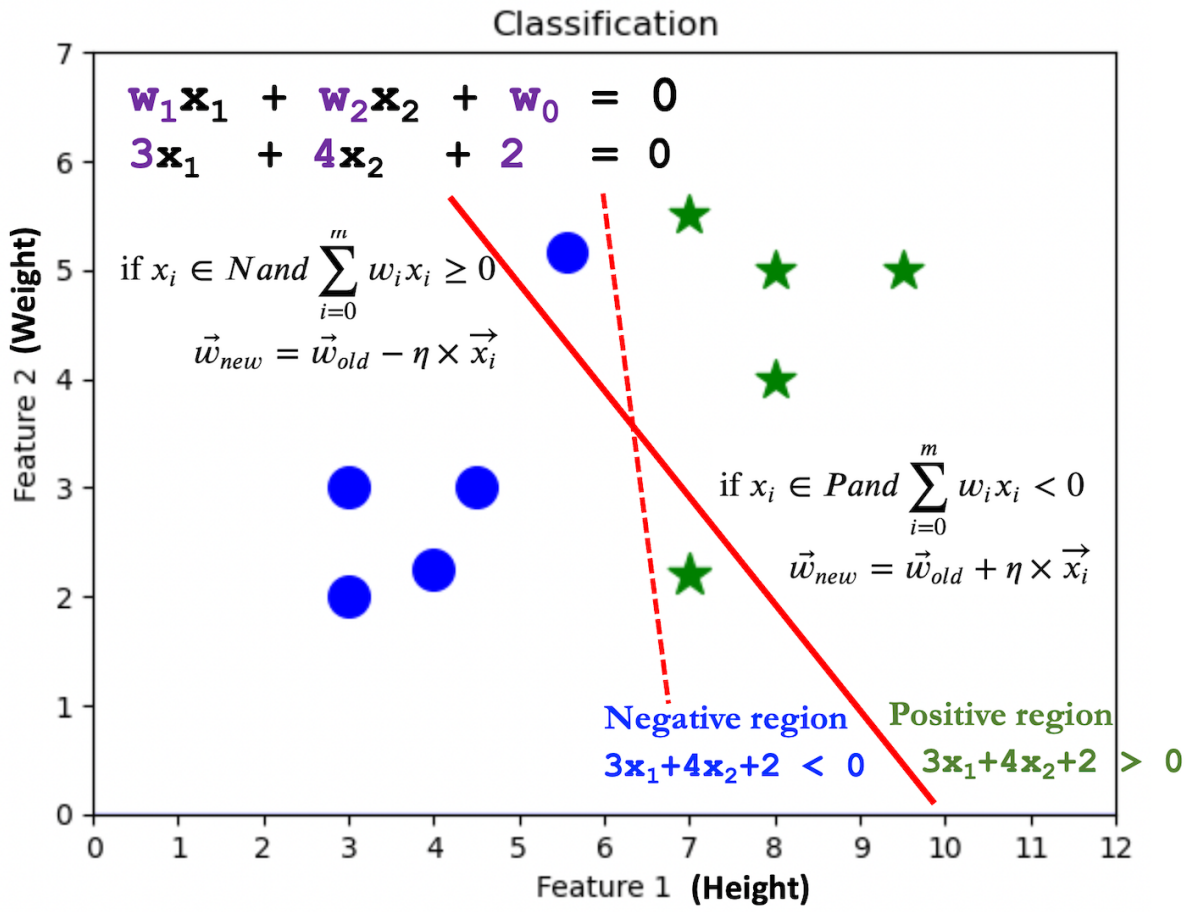
$$\vec{w}\vec{x} + w_0 = 0$$



https://www.desmos.com/calculator (https://www.desmos.com/calculator)

Pseudocode:

1. Initialize weights/coefficients with random values
2. Run a loop either a fixed number of times or till convergence
   - Select a random point
   - If (correctly classified)
     - Leave it
   - If (miss-classified)
     - Move the line towards the point



- If a `negative` point is in the `positive` region you `Subtract`
- If a `positive` point is in the `negative` region you `Add`

Algorithm 1:

1. epochs = 100, $\eta = 0.01$
2. for i in epochs:
3. Pick random $x_i \in P \cup N$
4. if $x_i \in N and \sum_{i=0}^{m} w_i x_i \geq 0$ then #A -ve point in +ve region
5. $\vec{w}_{new} = \vec{w}_{old} - \eta \times \vec{x_i}$
6. if $x_i \in P and \sum_{i=0}^{m} w_i x_i < 0$ then #A +ve point in -ve region

7. $\vec{w}_{new} = \vec{w}_{old} + \eta \times \vec{x_i}$

> - Note: In each iteration, we update the weights only if the randomly selected point is misclassified

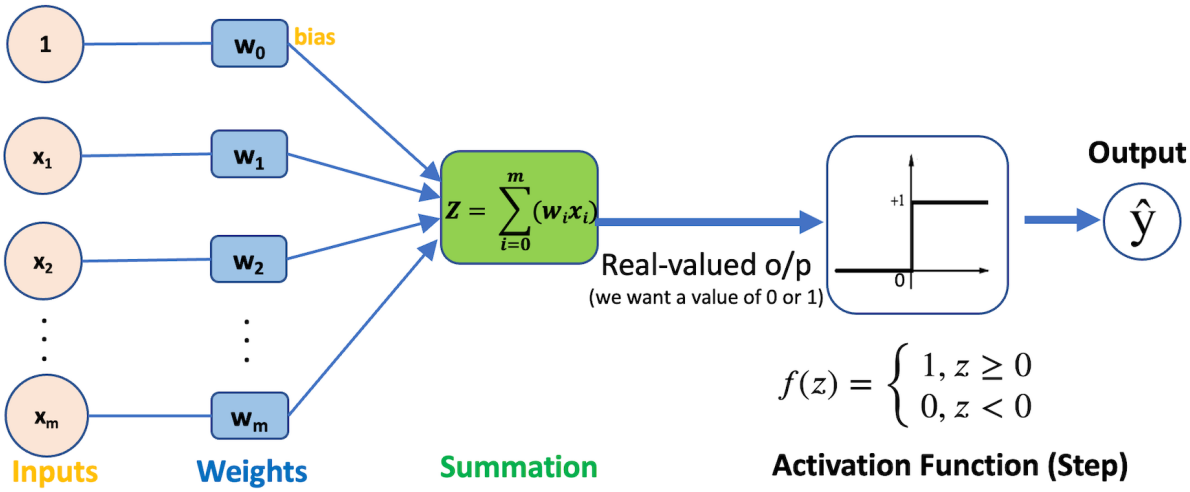
Algorithm 2: A more elegant way of writing above Algo without using the two if conditions

1. epochs = 100, $\eta = 0.01$
2. for i in epochs:
3. Pick random $x_i \in P \cup N$
4. $\vec{w}_{new} = \vec{w}_{old} + \eta \times (y_i - \hat{y}_i)\vec{x_i}$

Note:

- If the data point is correctly classified, then $(y_i - \hat{y}_i)$ will be zero and no update will occur
- If $y_i = 1$ and $\hat{y}_i = 0$, that means a positive point in negative region, so addition will be done automatically
- If $y_i = 0$ and $\hat{y}_i = 1$, that means a negative point in positive region, so subtraction will be done automatically

3. Logistic Regression using Single Layer Perceptron (`Step Function`)



Algorithm 2:
epochs = 100, $\eta = 0.01$
for i in epochs:
randomly select a point x_i
$\vec{w}_{new} = \vec{w}_{old} + \eta \times (y_i - \hat{y}_i)\vec{x_i}$

Generate Dataset

In [1]:
```
from sklearn.datasets import make_classification
import numpy as np
X, y = make_classification(n_samples=100,
                           n_features=2,
                           n_classes=2,
                           n_clusters_per_class=1,
                           class_sep=20,
                           random_state=41,
                           n_informative=1,
                           n_redundant=0,
                           hypercube=False)
X.shape, y.shape
```

Out[1]:
```
((100, 2), (100,))
```

In [2]:
```
X[0:5]
```

Out[2]:
```
array([[ 0.51123145, -0.11697552],
       [ 0.06316371, -0.73115232],
       [-0.0425064 , -0.7081059 ],
       [-3.2891569 , -2.01199214],
       [ 0.1111445 ,  1.63493163]])
```

In [3]:
```
y
```

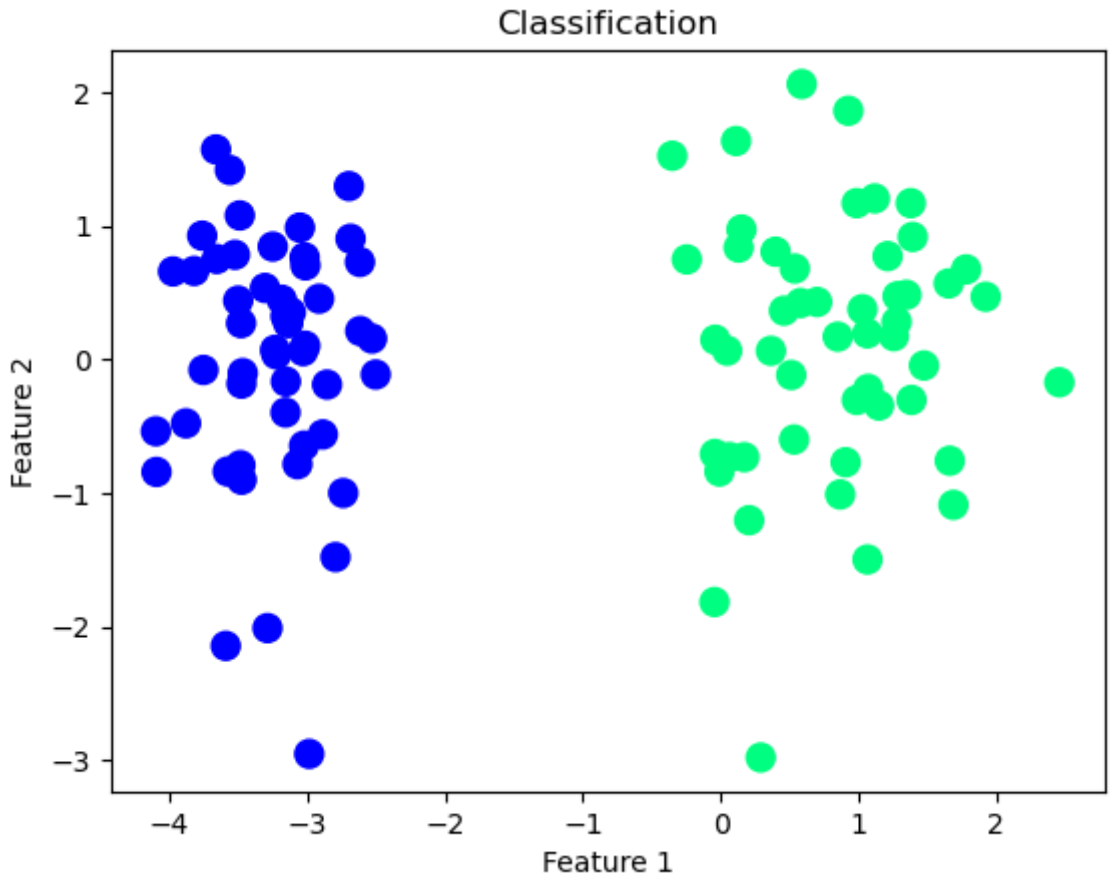
Out[3]:
```
array([1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0,
0,
       0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0,
1,
       0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1,
1,
       0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0,
0,
       1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1])
```

Visualize Dataset

In [4]:
```
import matplotlib.pyplot as plt

plt.scatter(x=X[:,0], y=X[:,1], c=y, cmap='winter',  s=100)

plt.title("Classification")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show();
```



Perceptron Training Algorithm

$$\vec{w}_{new} = \vec{w}_{old} + \eta \times (y_i - \hat{y}_i)\vec{x_i}$$

In [5]:
```
def perceptron(X,y):
    X = np.insert(X, 0, 1, axis=1)    # add an additional column of ones
    weights = np.ones(X.shape[1])     # initialize weights vector having
    lr = 0.1
    epocs = 1000
    for i in range(epocs):
        j = np.random.randint(0,100)                          # select a r
        y_hat = step( np.sum( np.dot(X[j], weights) ) )       # calculate
        weights = weights    +   lr * (y[j] - y_hat) * X[j]   # update the
    return weights[0],weights[1:]                             # return bia
```

In [6]:
```
def step(z):
    return 1 if z>=0 else 0
```

In [7]:
```
# Train the model and get the bias and weights
bias , weights = perceptron(X,y)
print("w0: ", bias)
print("w1 and w2: ", weights)
```

```
w0:  1.3000000000000003
w1 and w2:  [1.05343617 0.2221512 ]
```

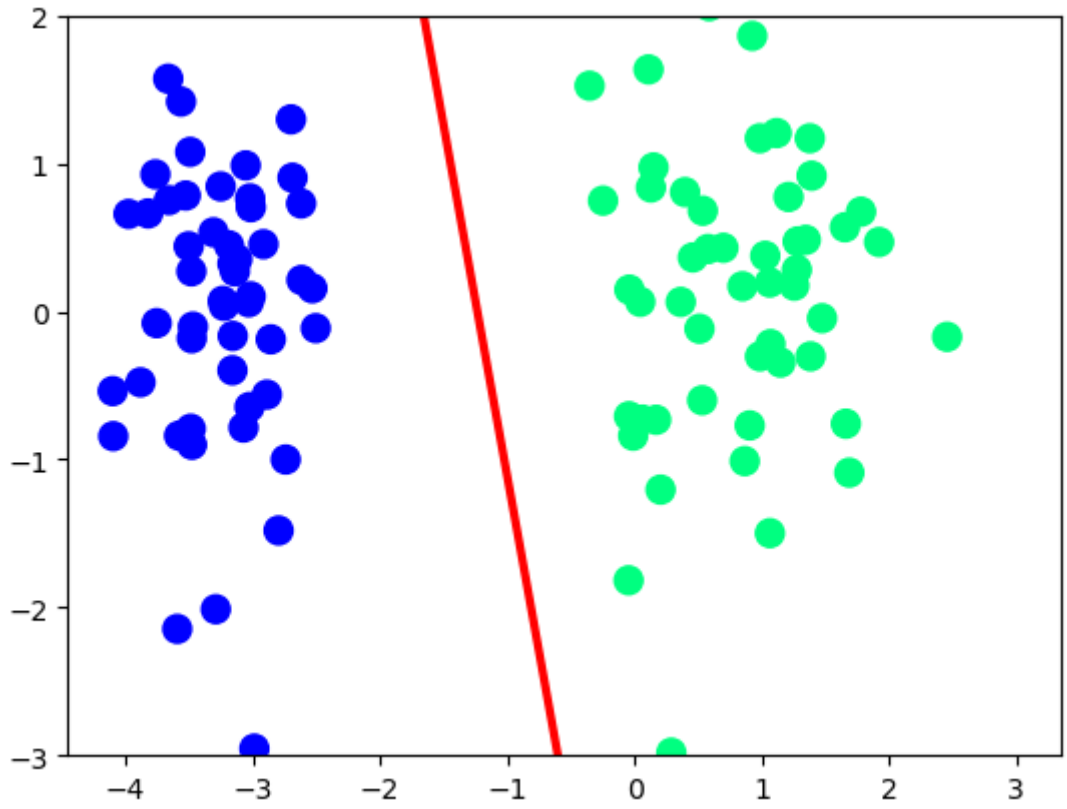
Visualize Result

- $Ax + By + C = 0$
- Slope $= -\frac{A}{B}$
- y-intercept $= -\frac{C}{B}$

In [8]:
```
m = -(weights[0]/weights[1])
b = -(bias/weights[1])

x_input = np.linspace(-3,3,100)
y_input = m*x_input + b

plt.plot(x_input, y_input, color='red', linewidth=3)
plt.scatter(X[:,0],X[:,1], c=y, cmap='winter',  s=100)
plt.ylim(-3,2)
plt.show();
```



4. Logistic Regression using Scikit-Learn Model

In [9]:
```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
model.fit(X, y)
```

Out[9]:
```
LogisticRegression()
```

In [10]:
```
print("w0: ", model.intercept_)
print("w1 and w2: ", model.coef_)
```

```
w0:  [3.13649441]
w1 and w2:  [[2.36687798 0.02178765]]
```

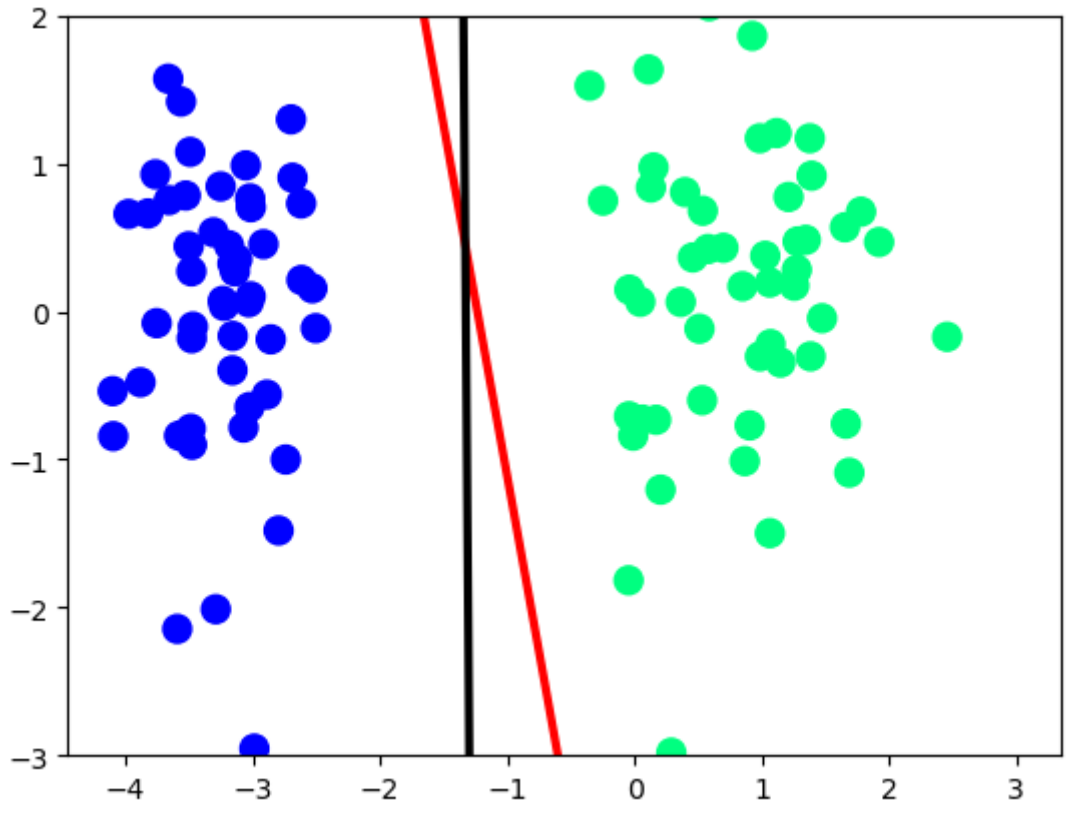
Calculating Slope and y-intercept:

- $Ax + By + C = 0$
- Slope $= -\frac{A}{B}$
- y-intercept $= -\frac{C}{B}$

In [11]:
```
m = -(model.coef_[0][0]/model.coef_[0][1])
b = -(model.intercept_/model.coef_[0][1])

x_input1 = np.linspace(-3,3,100)
y_input1 = m*x_input + b

plt.plot(x_input,  y_input,  color='red',   linewidth=3) # Resulting Lin
plt.plot(x_input1, y_input1, color='black', linewidth=3) # Resulting Lin
plt.scatter(X[:,0], X[:,1], c=y, cmap='winter', s=100)
plt.ylim(-3,2)
plt.show();
```



Visualize Decision Boundry using mlxtend Library:

In [ ]:
```
# https://rasbt.github.io/mlxtend/
#import sys
#!{sys.executable} -m pip install mlxtend --upgrade --no-deps
```

In [12]:
```
from mlxtend.plotting import plot_decision_regions

plot_decision_regions(X=X, y=y, clf=model)
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Logistic Regression Decsion Boundry')
plt.show();
```

5. Improving Single Layer Perceptron (`Step Function`) Algorithm

a. Limitation

Algorithm:

epochs = 100, $\eta = 0.01$

for i in epochs:

randomly select a point x_i

$\vec{w}_{new} = \vec{w}_{old} + \eta \times (y_i - \hat{y}_i)\vec{x_i}$

- Our perceptron algorithm work only till the time all data points are on the correct side of the line. As soon as it is done the algorithm stops. In simple words, no update occurs if the point picked at random is a correctly classified point, i.e., y and $\hat{y}$ are both either 0 or 1.
- We need to make changes in the update function:

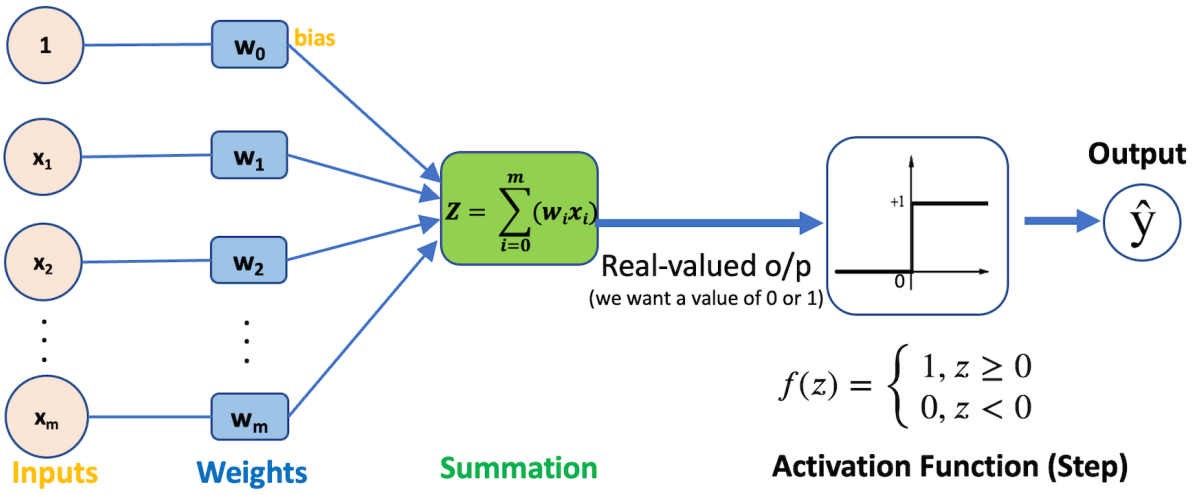
b. How to Overcome this Limitations?

An update should occur for both correctly as well as misclassified points

- **A `misclassified point pull` the line towards itself:**
  - If a point is `closer` to the line, it will pull the line with `smaller` magnitude
  - If a point is `far` from the line, it will pull the line with `greater` magnitude
- **A `correctly classified point push` the line away from it:**
  - If a point is `closer` to the line, it will push the line with `greater` magnitude
  - If a point is `far` from the line, it will push the line with `smaller` magnitude

c. How to Implement?

Old Technique (Using Step Function)



$$Z = \sum_{i=0}^{m} (w_i x_i)$$

$$f(z) = \begin{cases} 1, z \geq 0 \\ 0, z < 0 \end{cases}$$

$$\vec{w}_{new} = \vec{w}_{old} + \eta \times (y_i - \hat{y}_i)\vec{x_i}$$

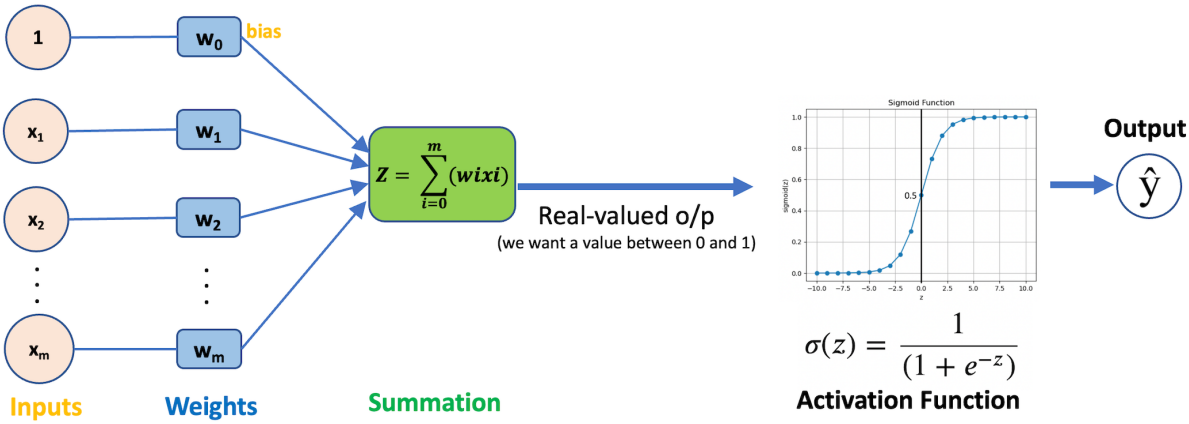
Limitation: Since the output of step function is discrete, i.e., either 0 or 1, therefore, the update of $\vec{w}$ occur only if a point is mis-classified

New Technique (Using Sigmoid Function)

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- The Sigmoid function takes any real value as input and outputs another value between 0 and 1 (probability).
- In Sigmoid function, e is the Euler's number, whose value is approximately 2.718, and is the base of the natural logarithms.
- The Sigmoid function is continuously differentiable, and monotonic function having a fixed output range.

$$\vec{w}_{new} = \vec{w}_{old} + \eta \times (y_i - \hat{y}_i)\overrightarrow{x_i}$$

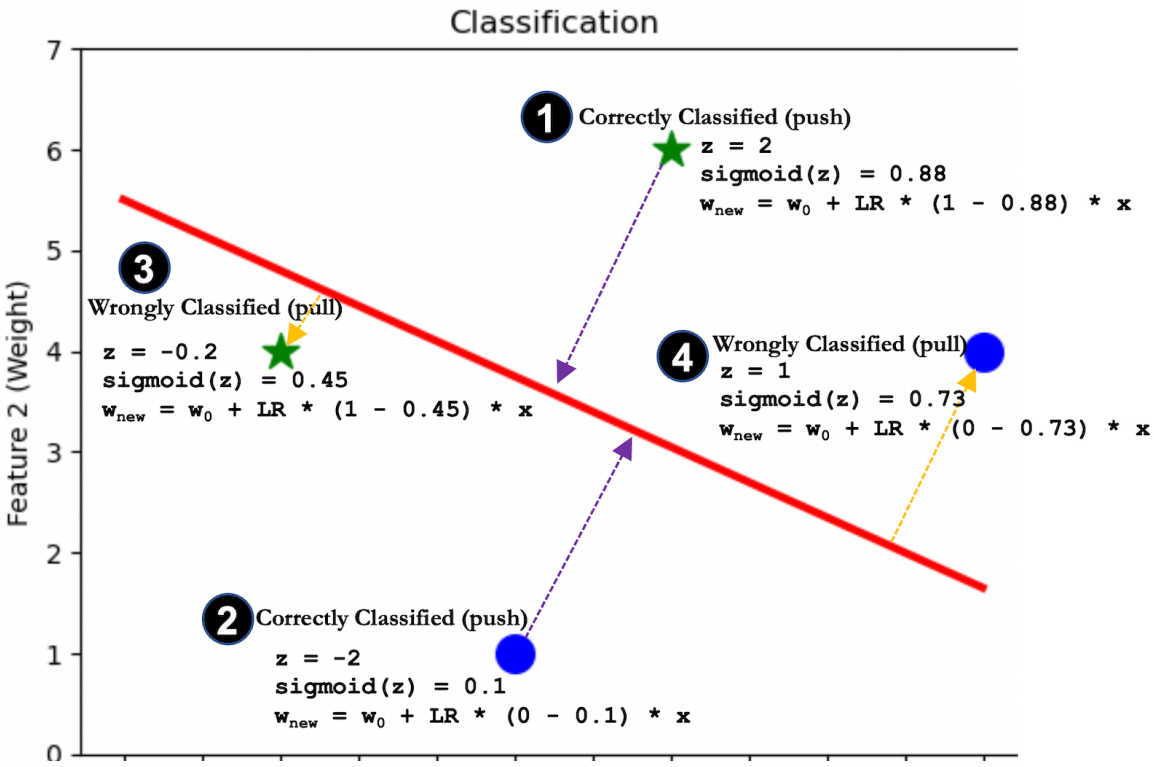


- If Z is a very large positive number, $\sigma(z)$ approaches to 1.0
- If Z is a very large negative number, $\sigma(z)$ approaches to 0.0
- If Z=0, $\sigma(z) = 0.5$
- By replacing the step function with sigmoid function, the value of $\hat{y}$ will not be discrete, rather will be in the range of 0 and 1.
- This will solve our problem, as the value of $(y_i - \hat{y_i})$ will not be zero even for the correctly classified points.
- So if we use sigmoid function, an update will occur in $\vec{w}$ for both correctly and incorrectly classified points. Because $\hat{y}$ is a continous value between 0 and 1. However the magnitude of update will be different.

Working on Sample Data Points:

- **A misclassified point `pull` the line towards itself**
- **A correctly classified point `push` the line away from it**

$$\vec{w}_{new} = \vec{w}_{old} + \eta \times (y_i - \hat{y}_i)\overrightarrow{x_i}$$



d. Python Code

Perceptron Training Algorithm using Sigmoid Function

$$\vec{w}_{new} = \vec{w}_{old} + \eta \times (y_i - \hat{y}_i)\vec{x_i}$$

In [13]:
```
def perceptron(X, y):
    X = np.insert(X, 0, 1, axis=1)     # add an additional column of ones
    weights = np.ones(X.shape[1])      # initialize weights vector having
    lr = 0.1
    epocs = 1000

    for i in range(epocs):
        j = np.random.randint(0,100)                          # select a r
        y_hat = sigmoid( np.sum( np.dot(X[j], weights) ) )    # calculate
        weights = weights  +  lr * (y[j] - y_hat) * X[j]      # update the
    return weights[0],weights[1:]                             # return bia
```

$$\sigma(z) = \frac{1}{(1+e^{-z})}$$

In [14]:
```
def sigmoid(z):
    return 1/(1 + np.exp(-z))
```

In [15]:
```
# Train the model and get the bias and weights
bias , weights = perceptron(X, y)
print("w0: ", bias)
print("w1 and w2: ", weights)
```

```
w0:  2.8448332374278924
w1 and w2:  [2.84092997 0.23470077]
```

Visualize Result

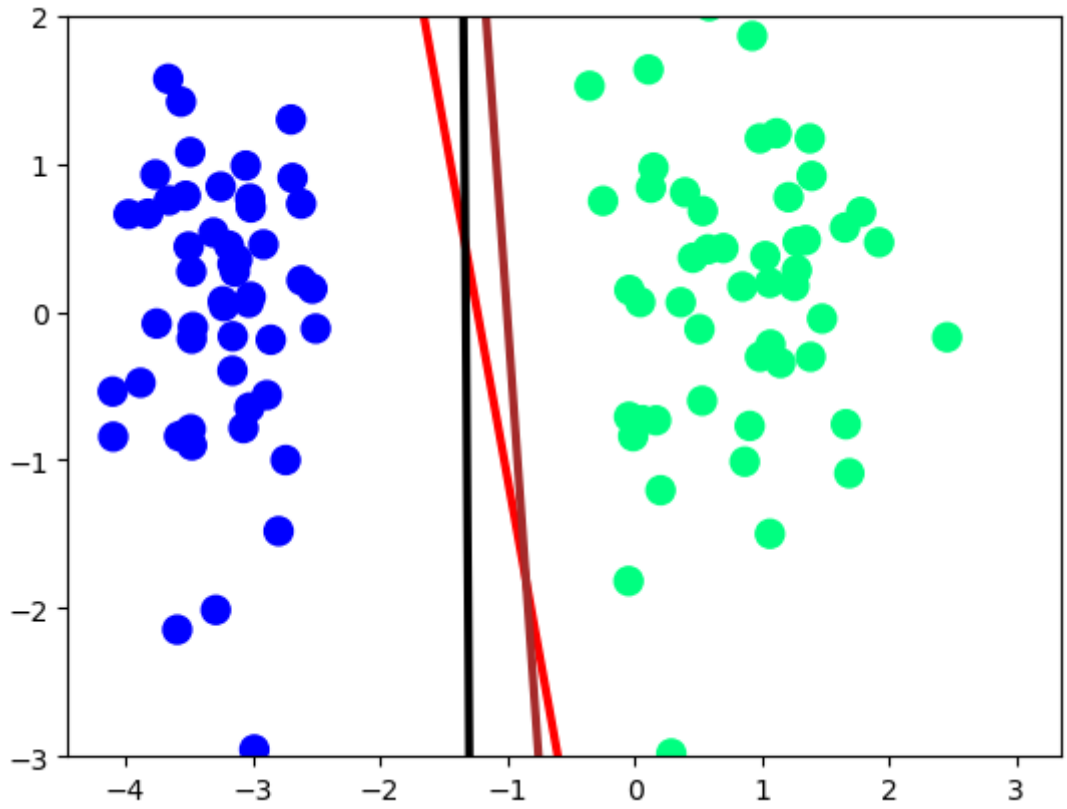
- $Ax + By + C = 0$
- Slope $= -\frac{A}{B}$
- y-intercept $= -\frac{C}{B}$

In [16]:
```
m = -(weights[0]/weights[1])
b = -(bias/weights[1])

x_input2 = np.linspace(-3,3,100)
y_input2 = m*x_input + b


plt.plot(x_input,y_input,color='red',linewidth=3)      # Resulting Line o
plt.plot(x_input1,y_input1,color='black',linewidth=3) # Resulting Line o
plt.plot(x_input2,y_input2,color='brown',linewidth=3) # Resulting Line o

plt.scatter(X[:,0],X[:,1],c=y,cmap='winter',s=100)
plt.ylim(-3,2)
plt.show();
```



- Each time you execute the perceptron algorithm using `step` or `sigmoid` function you may get different decision boundry, because you get different weights. The reason is random selection of data point in each iteration. Moreover, the resulting decision boundries are not as good as the one generated by Scikit-Learn's Logistic Regression Model
- The Scikit-Learn's Logistic Regression actually use `Gradient Descent` algorithm to find the optimal weights for a given dataset by minimizing the errors between the predicted probabilities and the true class labels by minimizing `Cross-Entropy-Loss` or `Log-Loss` function. More on this in next lecture...