National University of Computer & Emerging Sciences, Karachi
Computer Science Department

Spring 2022, Lab Manual – 09

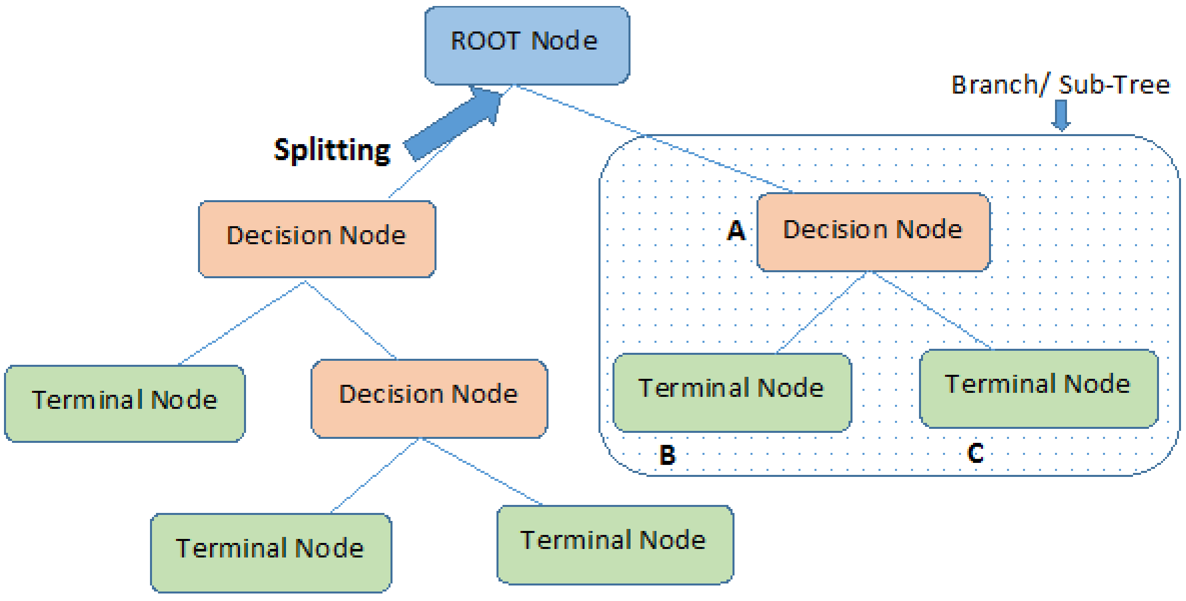
Course Code: AI-2002	**Course : Artificial Intelligence Lab**
Instructor(s):	**Kariz Kamal, Erum Shaheen, Mafaza Mohi, Danish Waseem, Ali Fatmi**

Contents:

Decision Tree

In general, Decision tree analysis is a predictive modeling tool that can be applied across many areas. Decision trees can be constructed by an algorithmic approach that can split the dataset in different ways based on different conditions. Decisions trees are one of the most powerful algorithms that falls under the category of supervised algorithm.

Below is an image explaining the basic structure of the decision tree. Every tree has a **root node**, where the inputs are passed through. This root node is further divided into sets of decision nodes where results and observations are conditionally based. The process of dividing a single node into multiple nodes is called **splitting**. If a node doesn't split into further nodes, then it's called a **leaf node**, or **terminal node**. A subsection of a decision tree is called a **branch** or **sub-tree** (e.g. in the box in the image below).



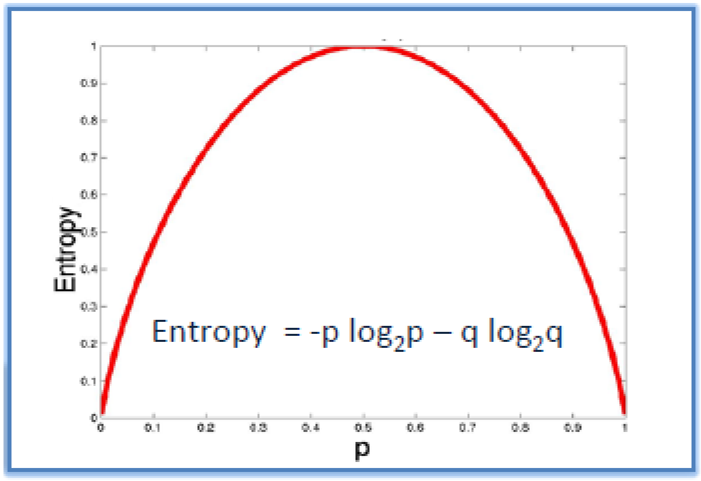
Note:- A is parent node of B and C.

Consider whether a dataset based on which we will determine whether to play football or not.

Outlook	Temperature	Humidity	Wind	Played football(yes/no)
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Overcast	Hot	High	Weak	Yes
Rain	Mild	High	Weak	Yes
Rain	Cool	Normal	Weak	Yes
Rain	Cool	Normal	Strong	No
Overcast	Cool	Normal	Strong	Yes
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Rain	Mild	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes
Rain	Mild	High	Strong	No

Entropy

In machine learning, entropy is a measure of the randomness in the information being processed. The higher the entropy, the harder it is to draw any conclusions from that information.

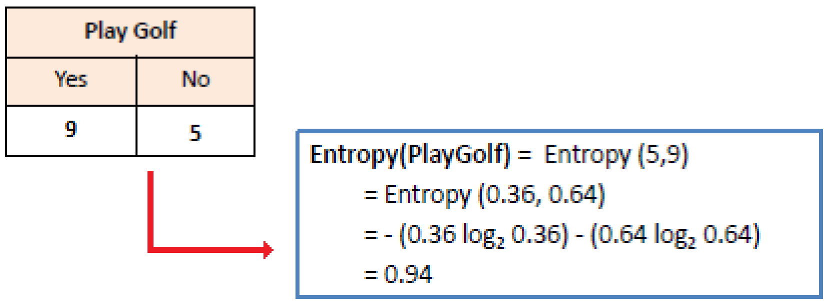


$$\text{Entropy} = -0.5 \log_2 0.5 - 0.5 \log_2 0.5 = 1$$

To build a decision tree, we need to calculate two types of entropy using frequency tables as follows:

a) Entropy using the frequency table of one attribute:

$$E(S) = \sum_{i=1}^{c} - p_i \log_2 p_i$$



Play Golf	
Yes	No
9	5

Entropy(PlayGolf) = Entropy (5,9)

= Entropy (0.36, 0.64)

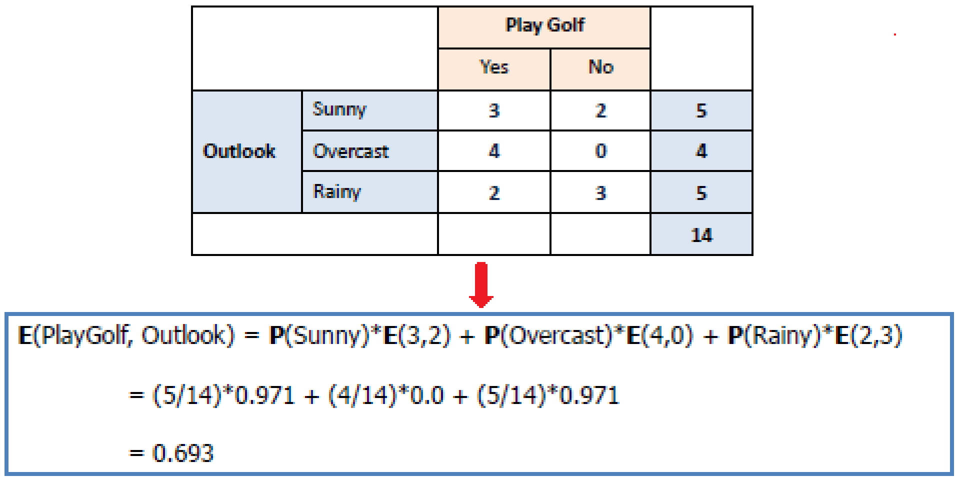
= - (0.36 $\log_2$ 0.36) - (0.64 $\log_2$ 0.64)

= 0.94

E(S) = -[(9/14)log(9/14) + (5/14)log(5/14)] = 0.94

b) Entropy using the frequency table of two attributes:

$$E(T,X) = \sum_{c \in X} P(c)E(c)$$



		Play Golf		
		Yes	No	
Outlook	Sunny	3	2	5
	Overcast	4	0	4
	Rainy	2	3	5
				14

E(PlayGolf, Outlook) = **P**(Sunny)***E**(3,2) + **P**(Overcast)***E**(4,0) + **P**(Rainy)***E**(2,3)

= (5/14)*0.971 + (4/14)*0.0 + (5/14)*0.971

= 0.693

We have to calculated average weighted entropy. ie, we have found the total of weights of each feature multiplied by probabilities.

E(S, outlook) = (5/14)*E(3,2) + (4/14)*E(4,0) + (5/14)*E(2,3) = (5/14)(-(3/5)log(3/5)-(2/5)log(2/5))+ (4/14)(0) + (5/14)((2/5)log(2/5)-(3/5)log(3/5)) = 0.693

Information Gain

Information gain can be defined as the amount of information gained about a random variable from observing another random variable. It can be considered as the difference between the entropy of parent node and weighted average entropy of child nodes. The information gain is based on the decrease in entropy after a dataset is split on an attribute. Constructing a decision tree is all about finding attribute that returns the highest information gain

Decision Tree Algorithm:

Step 1: Calculate entropy of the target.

$$\begin{aligned}\textbf{Entropy(PlayGolf)} &= \text{Entropy}(5,9)\\ &= \text{Entropy}(0.36, 0.64)\\ &= -(0.36 \log_2 0.36) - (0.64 \log_2 0.64)\\ &= 0.94\end{aligned}$$

Step 2: The dataset is then split on the different attributes. The entropy for each branch is calculated. Then it is added proportionally, to get total entropy for the split. The resulting entropy is subtracted from the entropy before the split. The result is the Information Gain, or decrease in entropy.

		Play Golf	
		Yes	No
Outlook	Sunny	3	2
	Overcast	4	0
	Rainy	2	3
Gain = 0.247			

		Play Golf	
		Yes	No
Temp.	Hot	2	2
	Mild	4	2
	Cool	3	1
Gain = 0.029			

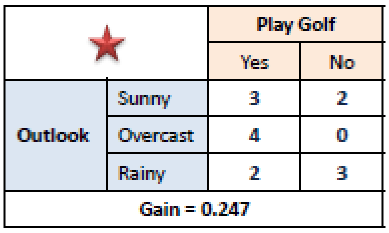
		Play Golf	
		Yes	No
Humidity	High	3	4
	Normal	6	1
Gain = 0.152			

		Play Golf	
		Yes	No
Windy	False	6	2
	True	3	3
Gain = 0.048			

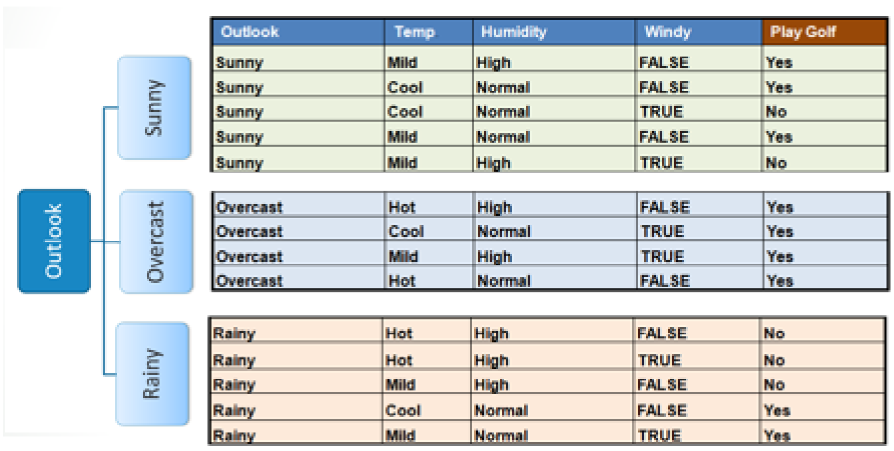
$$Gain(T,X) = Entropy(T) - Entropy(T,X)$$

$$\boxed{\begin{aligned}&\textbf{G}(\text{PlayGolf, Outlook}) = \textbf{E}(\text{PlayGolf}) - \textbf{E}(\text{PlayGolf, Outlook})\\ &\qquad = 0.940 - 0.693 = 0.247\end{aligned}}$$

Step 3: Choose attribute with the largest information gain as the decision node, divide the dataset by its branches and repeat the same process on every branch.

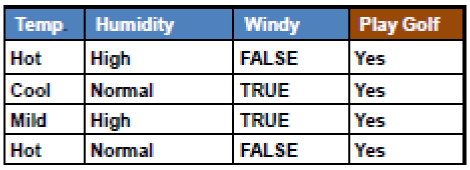


		Play Golf	
		Yes	No
Outlook	Sunny	3	2
	Overcast	4	0
	Rainy	2	3
Gain = 0.247			

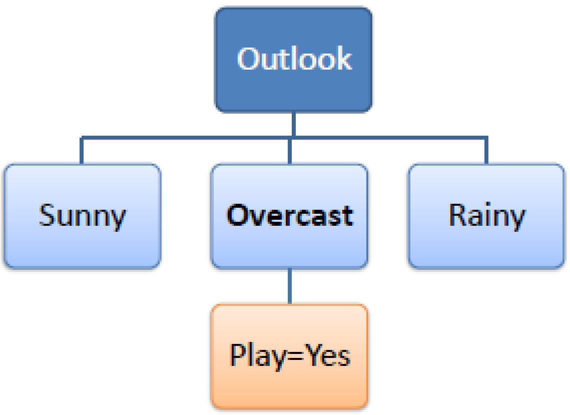


Outlook	Temp	Humidity	Windy	Play Golf
Sunny	Mild	High	FALSE	Yes
Sunny	Cool	Normal	FALSE	Yes
Sunny	Cool	Normal	TRUE	No
Sunny	Mild	Normal	FALSE	Yes
Sunny	Mild	High	TRUE	No
Overcast	Hot	High	FALSE	Yes
Overcast	Cool	Normal	TRUE	Yes
Overcast	Mild	High	TRUE	Yes
Overcast	Hot	Normal	FALSE	Yes
Rainy	Hot	High	FALSE	No
Rainy	Hot	High	TRUE	No
Rainy	Mild	High	FALSE	No
Rainy	Cool	Normal	FALSE	Yes
Rainy	Mild	Normal	TRUE	Yes

Step 4a: A branch with entropy of 0 is a leaf node.

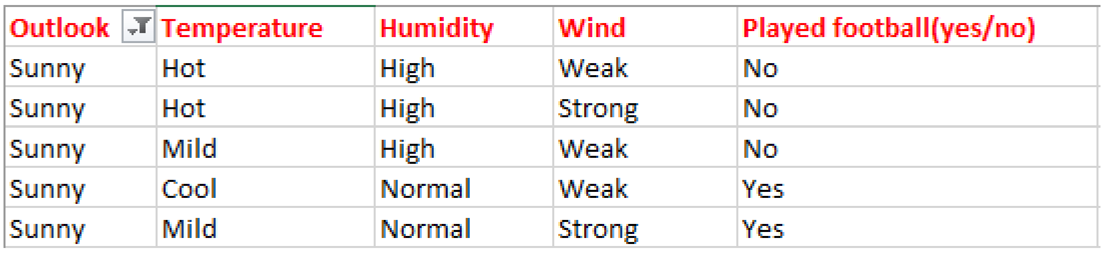


Temp	Humidity	Windy	Play Golf
Hot	High	FALSE	Yes
Cool	Normal	TRUE	Yes
Mild	High	TRUE	Yes
Hot	Normal	FALSE	Yes



Step 4b: A branch with entropy more than 0 needs further splitting.

The next step is to find the next node in our decision tree. Now we will find one under sunny. We have to determine which of the following Temperature, Humidity or Wind has higher information gain.



Outlook	Temperature	Humidity	Wind	Played football(yes/no)
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes

Calculate parent entropy E(sunny)

E(sunny) = (-(3/5)log(3/5)-(2/5)log(2/5)) = 0.971.

Now Calculate the information gain of Temperature. IG(sunny, Temperature)

		play		
		yes	no	total
	hot	0	2	2
Temperature	cool	1	1	2
	mild	1	0	1
				5

E(sunny, Temperature) = (2/5)*E(0,2) + (2/5)*E(1,1) + (1/5)*E(1,0)=2/5=0.4

Now calculate information gain.

IG(sunny, Temperature) = 0.971–0.4 =0.571

Similarly we get

IG(sunny, Humidity) = 0.971

IG(sunny, Windy) = 0.020

Here IG(sunny, Humidity) is the largest value. So Humidity is the node that comes under sunny.

	play	
Humidity	yes	no
high	0	3
normal	2	0

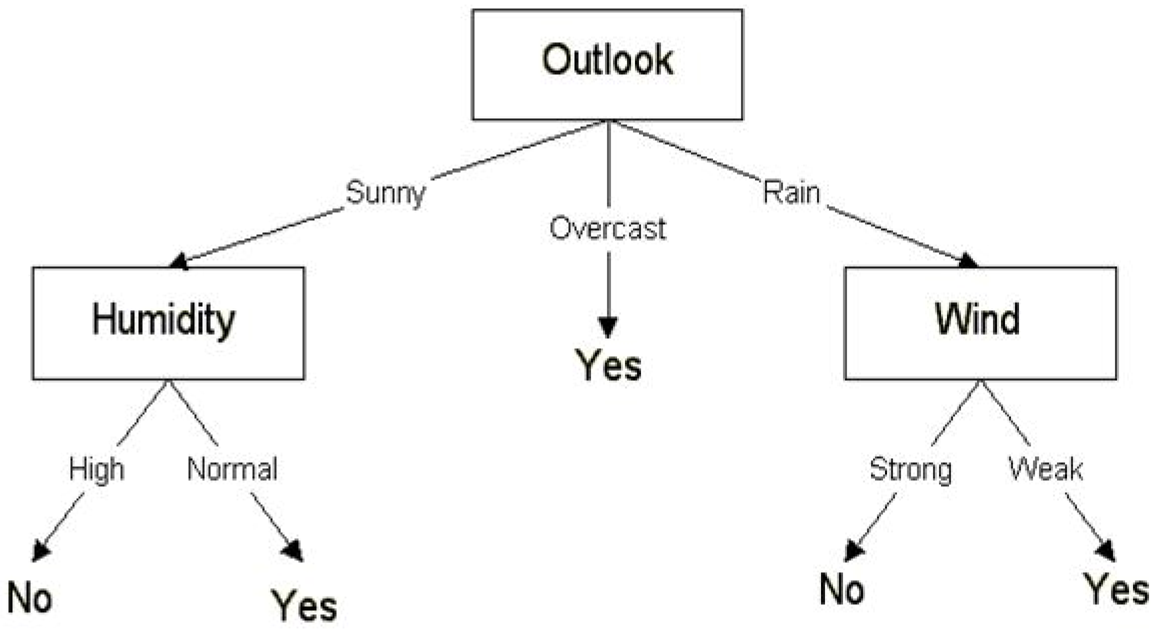
For humidity from the above table, we can say that play will occur if humidity is normal and will not occur if it is high. Similarly, find the nodes under rainy.

Step 5: The ID3 algorithm is run recursively on the non-leaf branches, until all data is classified.

Decision Tree to Decision Rules

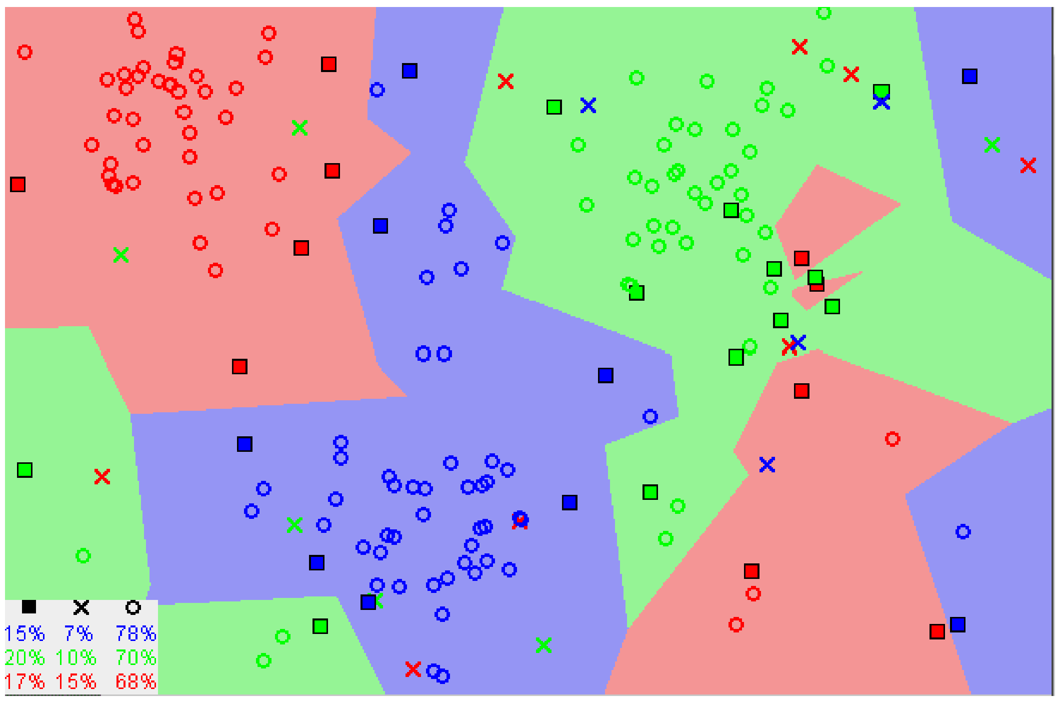
A decision tree can easily be transformed to a set of rules by mapping from the root node to the leaf nodes one by one.

Finally, our decision tree will look as below:



K-Nearest Classifier

The k-nearest neighbors (KNN) algorithm is a simple, easy-to-implement supervised machine learning algorithm that can be used to solve both classification and regression problems. The KNN algorithm assumes that similar things exist in close proximity. In other words, similar things are near to each other.



Notice in the image above that most of the time, similar data points are close to each other. The KNN algorithm hinges on this assumption being true enough for the algorithm to be useful. KNN captures the idea of similarity (sometimes called distance, proximity, or closeness) — calculating the distance between points on a graph (for example Euclidian distance).

How KNN algorithm works

Suppose we have height, weight and T-shirt size of some customers and we need to predict the T-shirt size of a new customer given only height and weight information we have. Data including height, weight and T-shirt size information is shown below –

Height (in cms)	Weight (in kgs)	T Shirt Size
158	58	M
158	59	M
158	63	M
160	59	M
160	60	M
163	60	M
163	61	M
160	64	L
163	64	L
165	61	L
165	62	L
165	65	L
168	62	L
168	63	L
168	66	L
170	63	L
170	64	L
170	68	L

Step 1: Calculate Similarity based on distance function

Euclidean :

$$d(x, y) = \sqrt{\sum_{i=1}^{m} (x_i - y_i)^2}$$

Manhattan / city - block :

$$d(x, y) = \sum_{i=1}^{m} |x_i - y_i|$$

The idea to use distance measure is to find the distance (similarity) between new sample and training cases and then finds the k-closest customers to new customer in terms of height and weight.

New customer has height 161cm and weight 61kg.

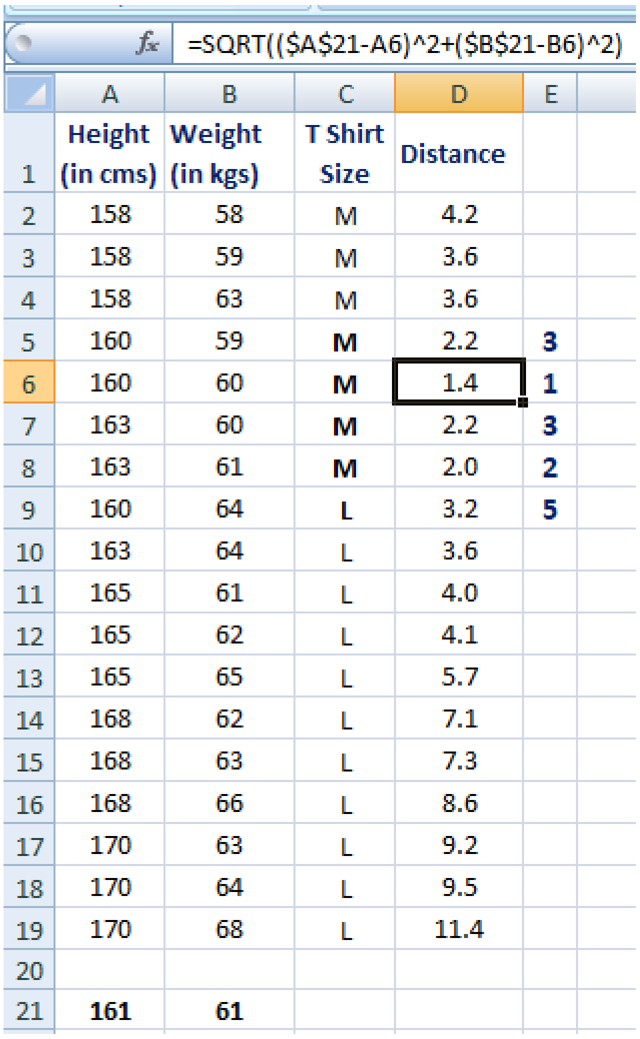
Euclidean distance between first observation and new observation is as follows –

=SQRT((161-158)^2+(61-58)^2)

Similarly, we will calculate distance of all the training cases with new case and calculates the rank in terms of distance. The smallest distance value will be ranked 1 and considered as nearest neighbor.

Step 2: Find K-Nearest Neighbors

Let k be 5. Then the algorithm searches for the 5 customers closest to Monica, i.e. most similar to Monica in terms of attributes, and see what categories those 5 customers were in. If 4 of them had ‘Medium T shirt sizes’ and 1 had ‘Large T shirt size’ then your best guess for Monica is ‘Medium T shirt. See the calculation shown in the snapshot below –

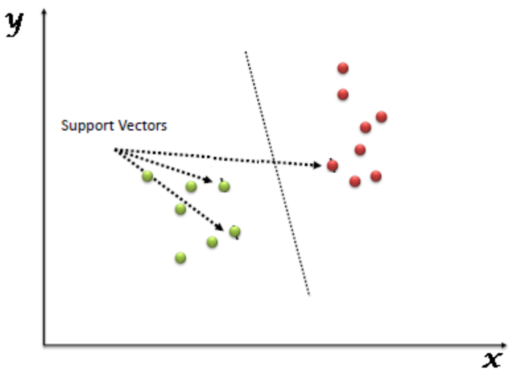


=SQRT((A21-A6)^2+(B21-B6)^2)

	A	B	C	D	E
1	**Height (in cms)**	**Weight (in kgs)**	**T Shirt Size**	**Distance**	
2	158	58	M	4.2	
3	158	59	M	3.6	
4	158	63	M	3.6	
5	160	59	**M**	2.2	**3**
6	160	60	**M**	1.4	**1**
7	163	60	**M**	2.2	**3**
8	163	61	**M**	2.0	**2**
9	160	64	**L**	3.2	**5**
10	163	64	L	3.6	
11	165	61	L	4.0	
12	165	62	L	4.1	
13	165	65	L	5.7	
14	168	62	L	7.1	
15	168	63	L	7.3	
16	168	66	L	8.6	
17	170	63	L	9.2	
18	170	64	L	9.5	
19	170	68	L	11.4	
20					
21	**161**	**61**			

SVM:

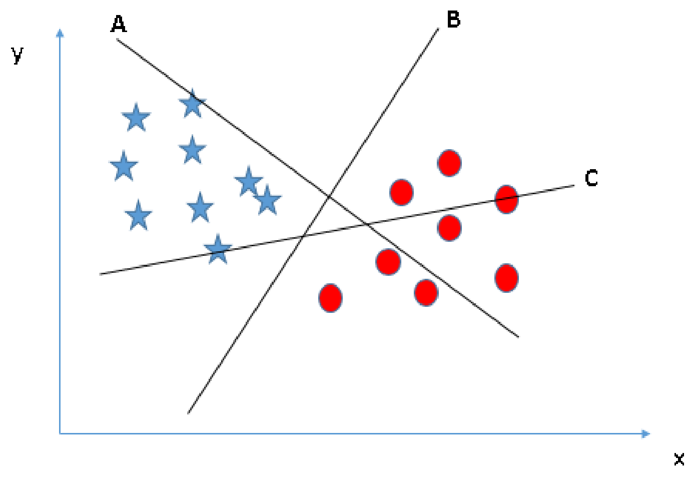
Support Vector Machine (SVM) is a supervised machine learning algorithm that can be used for both classification and regression challenges. However, it is mostly used in classification problems. In the SVM algorithm, we plot each data item as a point in n-dimensional space (where n is a number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane that differentiates the two classes very well (look at the below snapshot).



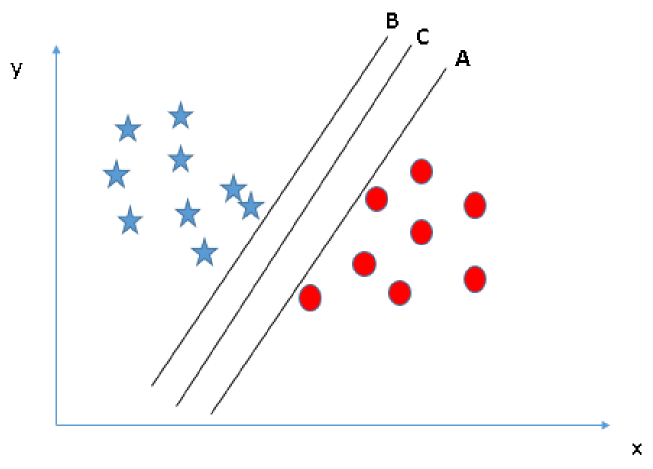
Support Vectors are simply the coordinates of individual observation. The SVM classifier is a frontier that best segregates the two classes (hyper-plane/ line).

How does SVM work

- **Identify the right hyper-plane (Scenario-1):** Here, we have three hyper-planes (A, B, and C). Now, identify the right hyper-plane to classify stars and circles.

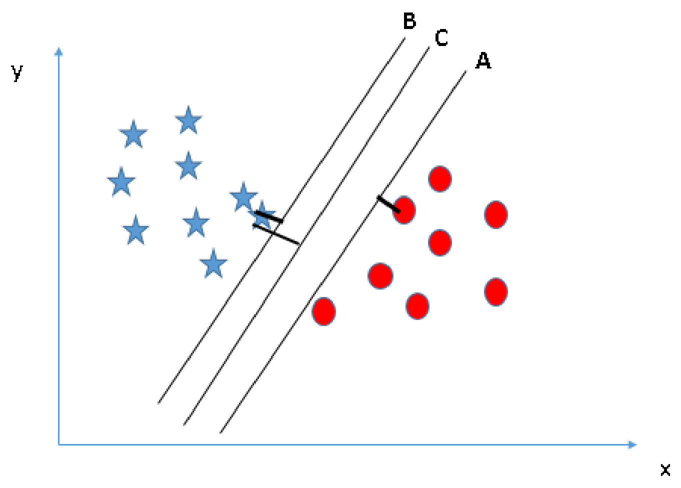


- You need to remember a thumb rule to identify the right hyper-plane: "Select the hyper-plane which segregates the two classes better". In this scenario, hyper-plane "B" has excellently performed this job.
- **Identify the right hyper-plane (Scenario-2):** Here, we have three hyper-planes (A, B, and C) and all are segregating the classes well. Now, how can we identify the right hyper-plane?



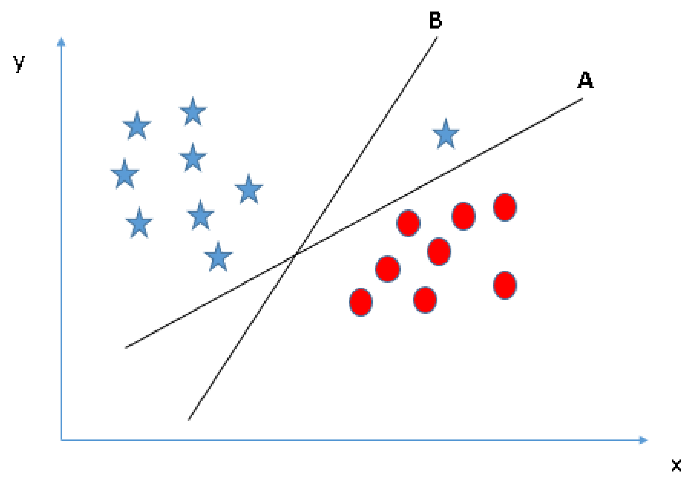
Here, maximizing the distances between nearest data point (either class) and hyper-plane

will help us to decide the right hyper-plane. This distance is called as **Margin**.



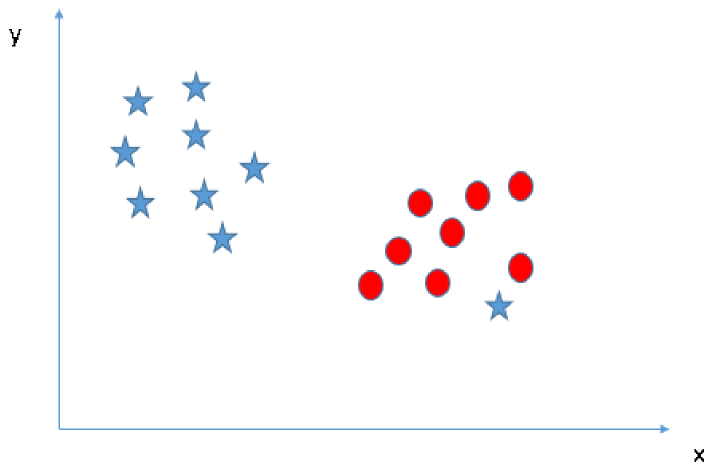
Above, you can see that the margin for hyper-plane C is high as compared to both A and B. Hence, we name the right hyper-plane as C. Another lightning reason for selecting the hyper-plane with higher margin is robustness. If we select a hyper-plane having low margin then there is high chance of miss-classification.

- **Identify the right hyper-plane (Scenario-3):**

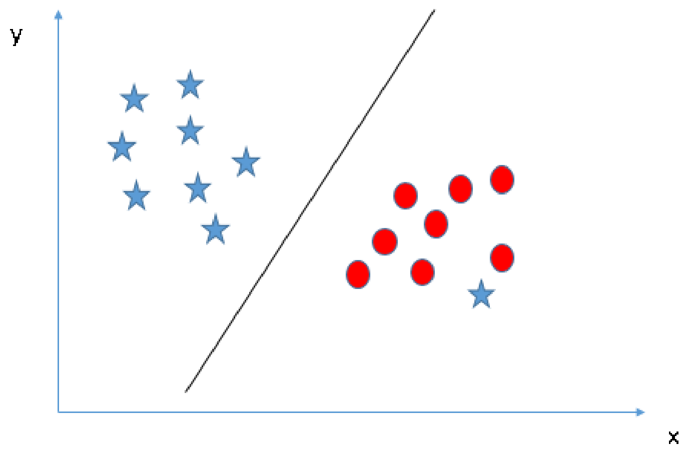


SVM selects the hyper-plane which classifies the classes accurately prior to maximizing margin. Here, hyper-plane B has a classification error and A has classified all correctly. Therefore, the right hyper-plane is **A.**

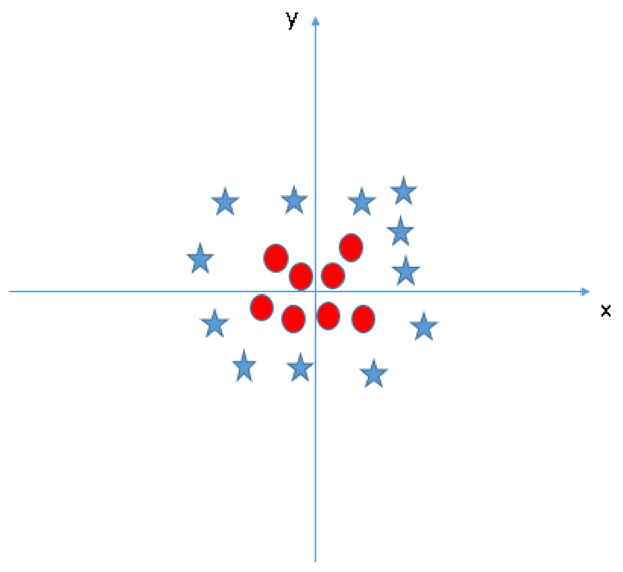
- **Can we classify two classes (Scenario-4)?:** Below, we are unable to segregate the two classes using a straight line, as one of the stars lies in the territory of other(circle) class as an outlier.



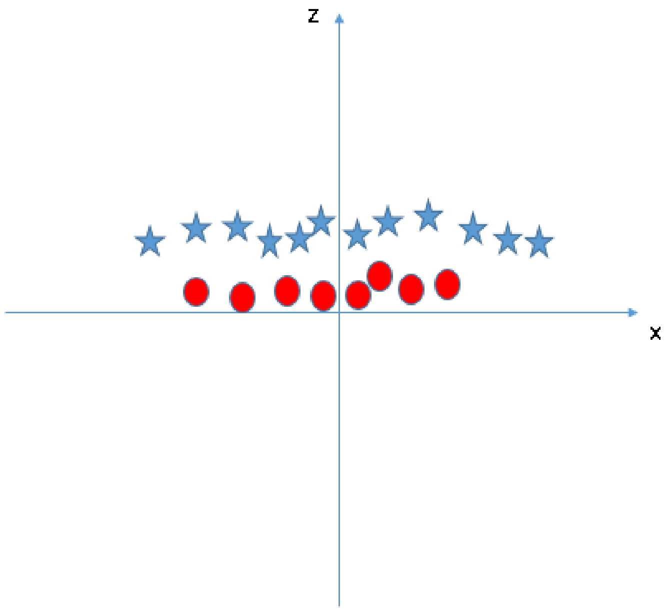
- One star at other end is like an outlier for star class. The SVM algorithm has a feature to ignore outliers and find the hyper-plane that has the maximum margin. Hence, we can say, SVM classification is robust to outliers.



- **Find the hyper-plane to segregate to classes (Scenario-5):** In the scenario below, we can't have linear hyper-plane between the two classes, so how does SVM classify these two classes? Till now, we have only looked at the linear hyper-plane.

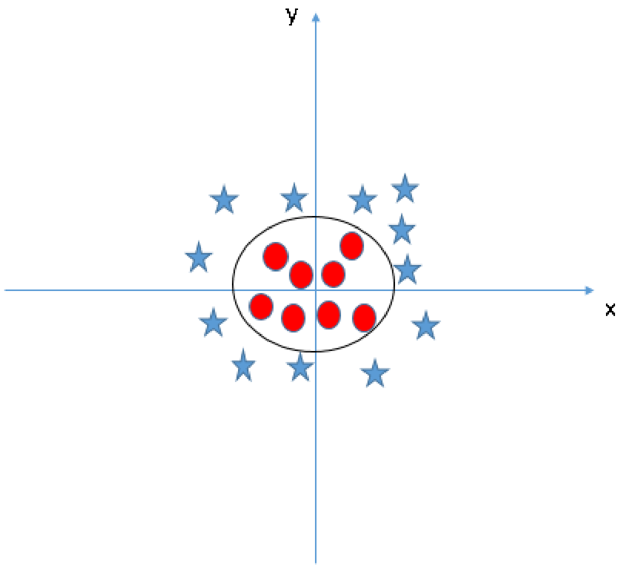


- SVM can solve this problem. Easily! It solves this problem by introducing additional feature. Here, we will add a new feature z=x^2+y^2. Now, let's plot the data points on axis x and z:



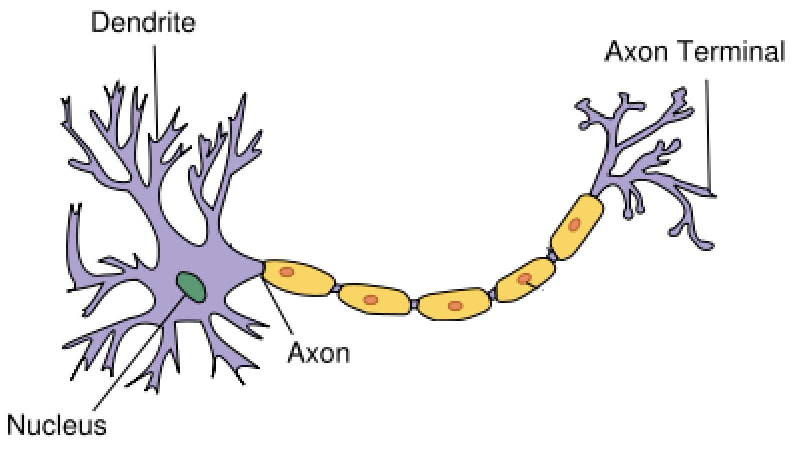
In the SVM classifier, it is easy to have a linear hyper-plane between these two classes. But, should we need to add this feature manually to have a hyper-plane? No, the SVM algorithm has a technique called the kernel **trick**. The SVM kernel is a function that takes low dimensional input space and transforms it to a higher dimensional space i.e. it converts not separable problem to separable problem. It is mostly useful in non-linear separation problem. Simply put, it does some extremely complex data transformations, then finds out the process to separate the data based on the labels or outputs you've defined.

When we look at the hyper-plane in original input space it looks like a circle:



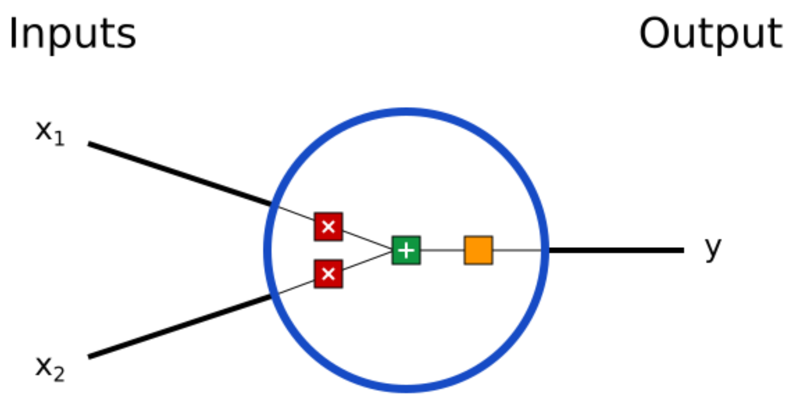
Simple Neural Network:

Artificial Neural Network ANN is an efficient computing system whose central theme is borrowed from the analogy of biological neural networks. ANN acquires a large collection of units that are interconnected in some pattern to allow communication between the units. These units, referred to as neurons, are simple processors which operate in parallel.



Neurons:

Neuron is the basic unit of a neural network. **A neuron takes inputs, does some math with them, and produces one output**. Here's what a 2-input neuron looks like:



3 things are happening here. First, each input is multiplied by a weight:

$$x_1 \rightarrow x_1 * w_1$$

$$x_2 \rightarrow x_2 * w_2$$

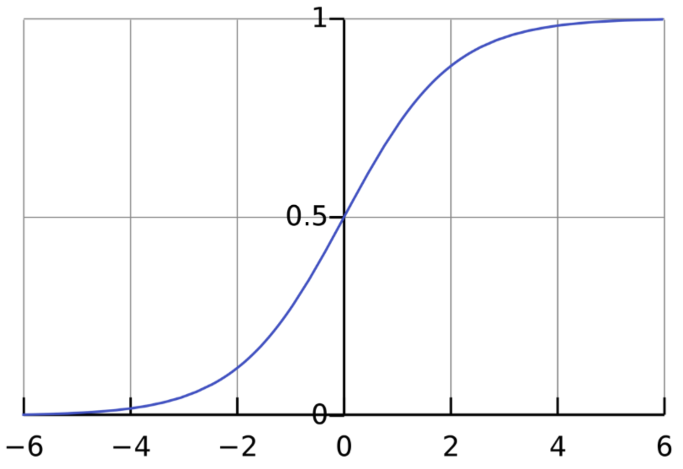
Next, all the weighted inputs are added together with a bias b:

$$(x_1 * w_1) + (x_2 * w_2) + b$$

Finally, the sum is passed through an activation function:

$$y = f(x_1 * w_1 + x_2 * w_2 + b)$$

The **activation function** is used to turn an unbounded input into an output that has a nice, predictable form. A commonly used activation function is the **sigmoid function**:



The sigmoid function only outputs numbers in the range (0,1). You can think of it as compressing (−∞,+∞) to (0,1) — big negative numbers become ~0, and big positive numbers become ~1.

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

A Simple Example

Assume we have a 2-input neuron that uses the sigmoid activation function and has the following parameters:

$$w = [0, 1]$$

$$b = 4$$

w=[0, 1] is just a way of writing w1=0, w2=1 in vector form.

Now, let's give the neuron an input of x=[2, 3]. We'll use the dot product to write things more concisely:

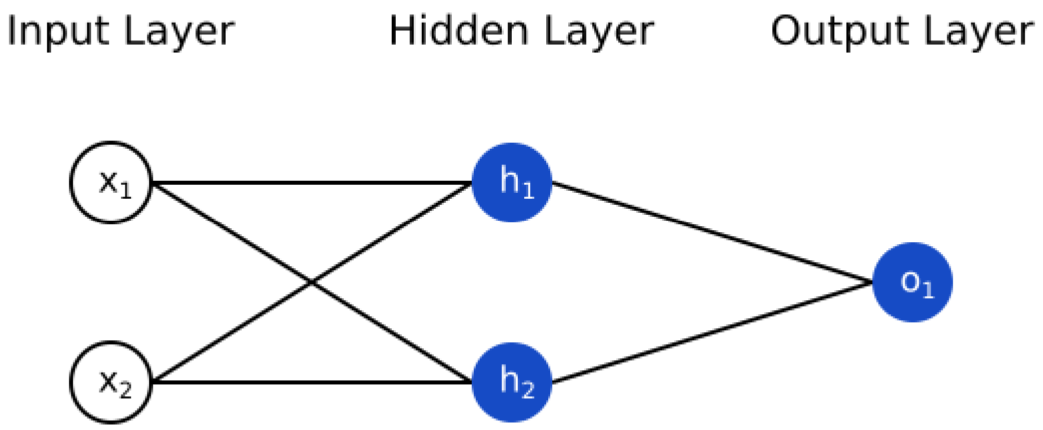
$$\begin{aligned}(w \cdot x) + b &= ((w_1 * x_1) + (w_2 * x_2)) + b \\ &= 0 * 2 + 1 * 3 + 4 \\ &= 7\end{aligned}$$

$$y = f(w \cdot x + b) = f(7) = \boxed{0.999}$$

The neuron outputs 0.999 given the inputs x=[2,3]. That's it! This process of passing inputs forward to get an output is known as **feed forward**.

Combining Neurons into a Neural Network

A neural network is nothing more than a bunch of neurons connected together. Here's what a simple neural network might look like:



This network has 2 inputs, a hidden layer with 2 neurons (*h*1 and *h*2), and an output layer with 1 neuron (*o*1). Notice that the inputs for *o*1 are the outputs from *h*1 and *h*2 — that's what makes this a network. A hidden layer is any layer between the input (first) layer and output (last) layer. There can be multiple hidden layers.

An Example: Feedforward

Let’s use the network pictured above and assume all neurons have the same weights *w*=[0,1], the same bias *b*=0, and the same sigmoid activation function. Let *h*1, *h*2 , *o*1 denote the *outputs* of the neurons they represent.

What happens if we pass in the input x=[2, 3]?

$$\begin{aligned} h_1 = h_2 &= f(w \cdot x + b) \\ &= f((0 * 2) + (1 * 3) + 0) \\ &= f(3) \\ &= 0.9526 \end{aligned}$$

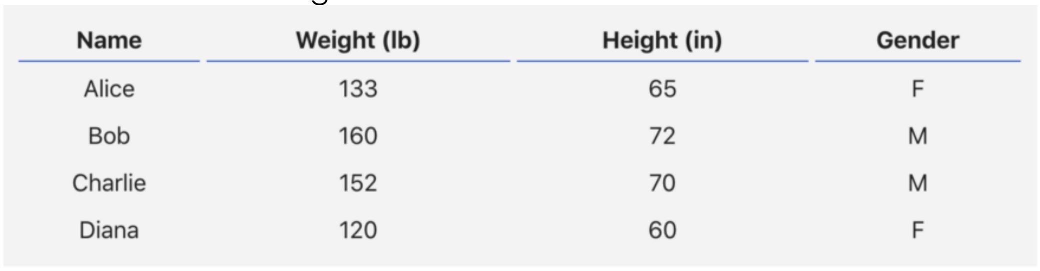
$$\begin{aligned} o_1 &= f(w \cdot [h_1, h_2] + b) \\ &= f((0 * h_1) + (1 * h_2) + 0) \\ &= f(0.9526) \\ &= \boxed{0.7216} \end{aligned}$$

The output of the neural network for input x=[2,3] is 0.7216. Pretty simple, right?

A neural network can have **any number of layers** with **any number of neurons** in those layers. The basic idea stays the same: feed the input(s) forward through the neurons in the network to get the output(s) at the end.

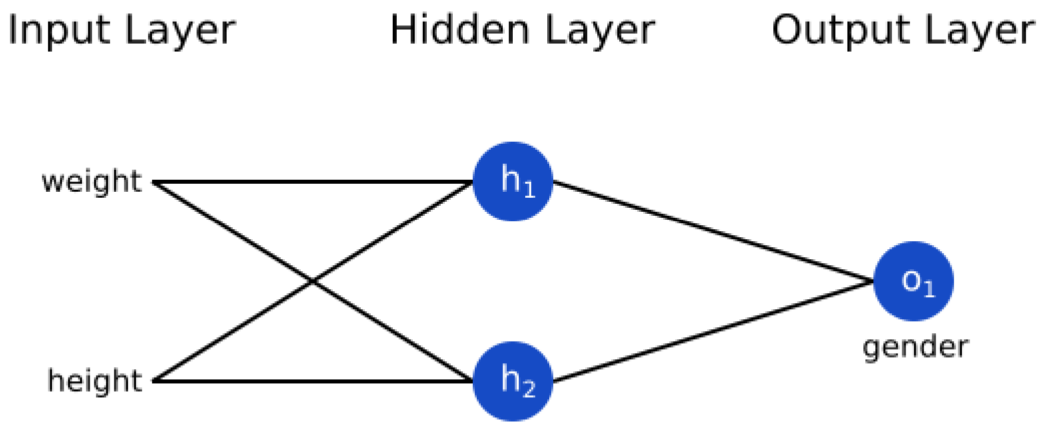
Training a Neural Network

Say we have the following measurements:



Name	Weight (lb)	Height (in)	Gender
Alice	133	65	F
Bob	160	72	M
Charlie	152	70	M
Diana	120	60	F

Let's train our network to predict someone's gender given their weight and height:



We'll represent Male with a 0 and Female with a 1, and we'll also shift the data to make it easier to use:

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1
Bob	25	6	0
Charlie	17	4	0
Diana	-15	-6	1

Loss

Before we train our network, we first need a way to quantify how "good" it's doing so that it can try to do "better". That's what the **loss** is.

We'll use the **mean squared error** (MSE) loss:

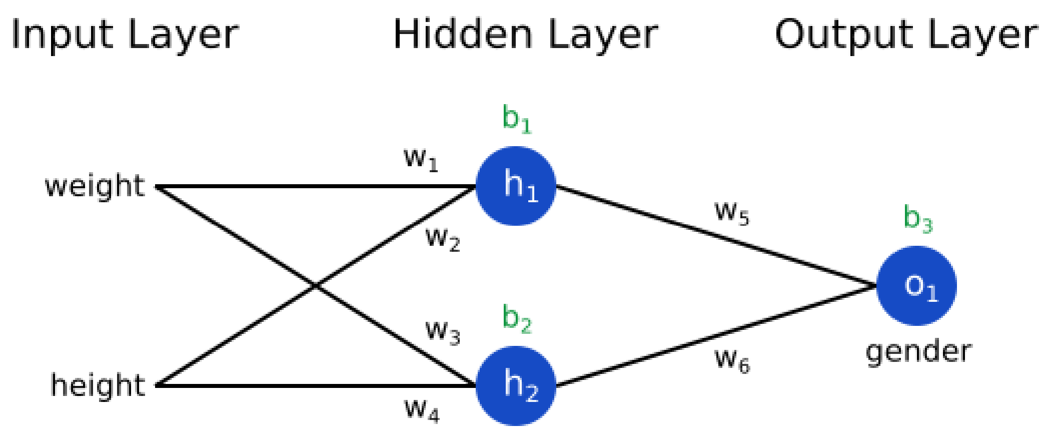
$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})^2$$

Our loss function is simply taking the average over all squared errors (hence the name *mean* squared error). The better our predictions are, the lower our loss will be!

Better predictions = Lower loss.

Training a network = trying to minimize its loss.

We now have a clear goal: **minimize the loss** of the neural network. We know we can change the network's weights and biases to influence its predictions, but how do we do so in a way that decreases loss?



Imagine we wanted to tweak *w*1. How would loss *L* change if we changed *w*1? That's a question the partial derivative can answer.

Chain Rule.

$$\boxed{\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}}$$

This system of calculating partial derivatives by working backwards is known as **backpropagation**, or "backprop".

Stochastic Gradient Descent

We have all the tools we need to train a neural network now! We'll use an optimization algorithm called stochastic gradient descent (SGD) that tells us how to change our weights and biases to minimize loss. It's basically just this update equation:

$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

η is a constant called the **learning rate** that controls how fast we train. All we're doing is subtracting *η ∂w*1/*∂L* from *w*1:

If we do this for every weight and bias in the network, the loss will slowly decrease and our network will improve.

Our training process will look like this:

1. Choose **one** sample from our dataset. This is what makes it *stochastic* gradient descent — we only operate on one sample at a time.
2. Calculate all the partial derivatives of loss with respect to weights or biases (e.g. *∂L*/*∂w*1 , *∂L*/*∂w*2, etc).
3. Use the update equation to update each weight and bias.
4. Go back to step 1.

TASKS: