Chapter No.4 Advance Classification

Radial Basis Function Kernel – Machine Learning

Radial Basis Kernel is a kernel function that is used in machine learning to find a non-linear classifier or regression line.

What is Kernel Function?
Kernel Function is used to transform n-dimensional input to m-dimensional input, where m is much higher than n then find the dot product in higher dimensional efficiently. The main idea to use kernel is: A linear classifier or regression curve in higher dimensions becomes a Non-linear classifier or regression curve in lower dimensions.

Mathematical Definition of Radial Basis Kernel:

$$K(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2}\right)$$

Radial Basis Kernel

where *x*, *x'* are vector point in any fixed dimensional space.

But if we expand the above exponential expression, It will go upto infinite power of *x* and *x'*, as expansion of e^x contains infinite terms upto infinite power of *x* hence it involves terms upto infinite powers in infinite dimension.

If we apply any of the algorithms like perceptron Algorithm or linear regression on this kernel, actually we would be applying our algorithm to new infinite-dimensional datapoint we have created. Hence it will give a hyperplane in infinite dimensions, which will give a very strong non-linear classifier or regression curve after returning to our original dimensions.

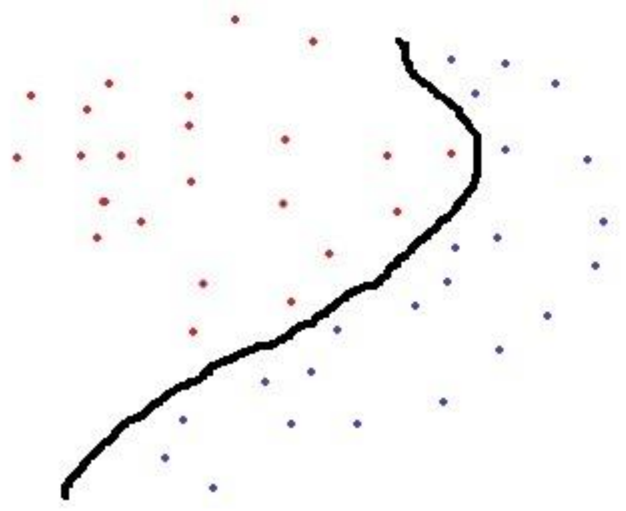
$$a_1 X^{inf} + a_2 X^{inf-1} + a_3 X^{inf-2} + \dots + a_{inf} X + C$$

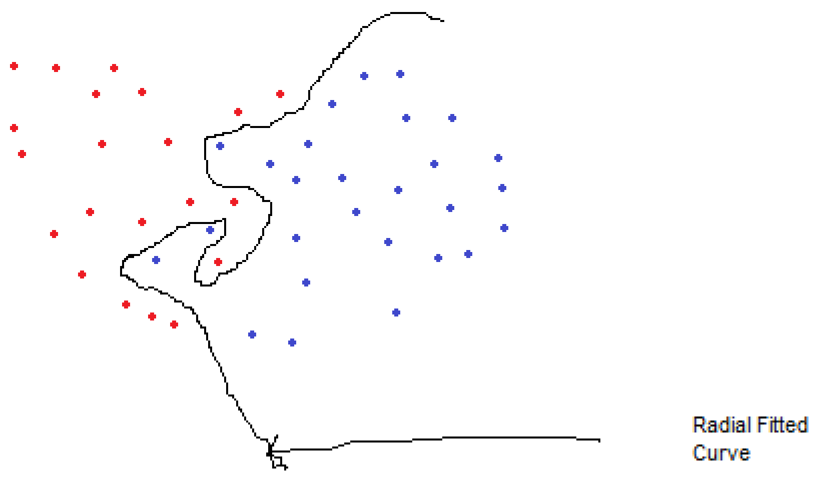
polynomial of infinite power

So, Although we are applying linear classifier/regression it will give a non-linear classifier or regression line, that will be a polynomial of infinite power. And being a polynomial of infinite power, Radial Basis kernel is a very powerful kernel, which can give a curve fitting any complex dataset.

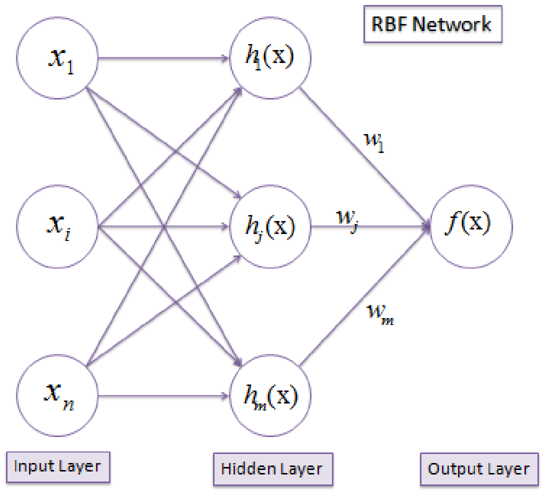
Why Radial Basis Kernel Is much powerful?
The main motive of the kernel is to do calculations in any d-dimensional space where *d > 1*, so that we can get a quadratic, cubic or any polynomial equation of large degree for our classification/regression line. Since Radial basis kernel uses exponent and as we know the expansion of e^x gives a polynomial equation of infinite power, so using this kernel, we make our regression/classification line infinitely powerful too.

Some Complex Dataset Fitted Using RBF Kernel easily:





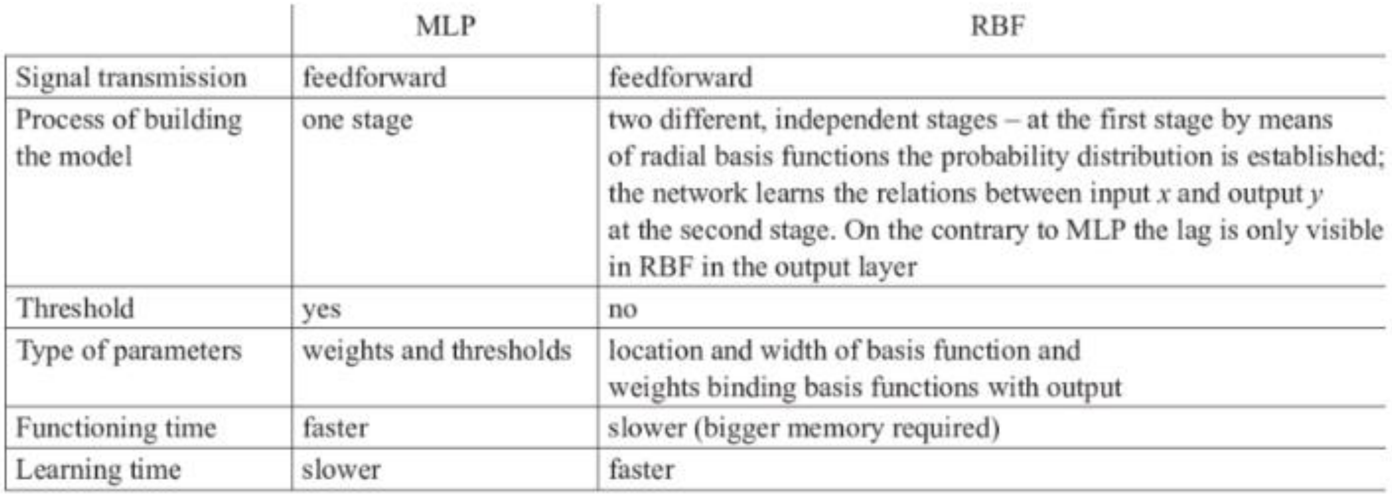
Architecture of RBF network:



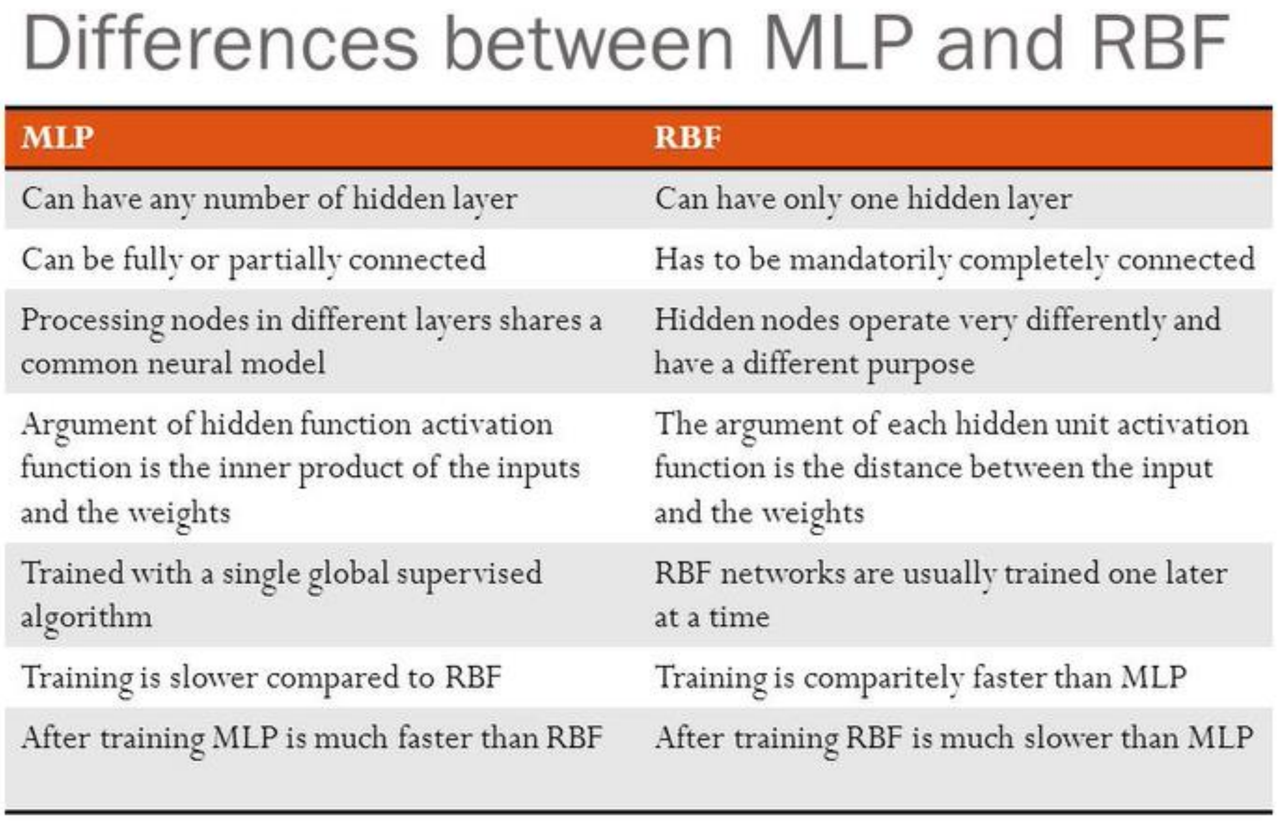
$$f(x) = \sum_{j=1}^{m} w_j h_j(x)$$

$$h(x) = \exp\left(- \frac{(x-c)^2}{r^2} \right)$$

Classification only happens on the second phase, where linear combination of hidden functions are driven to output layer.



	MLP	RBF
Signal transmission	feedforward	feedforward
Process of building the model	one stage	two different, independent stages – at the first stage by means of radial basis functions the probability distribution is established; the network learns the relations between input *x* and output *y* at the second stage. On the contrary to MLP the lag is only visible in RBF in the output layer
Threshold	yes	no
Type of parameters	weights and thresholds	location and width of basis function and weights binding basis functions with output
Functioning time	faster	slower (bigger memory required)
Learning time	slower	faster



Differences between MLP and RBF

MLP	RBF
Can have any number of hidden layer	Can have only one hidden layer
Can be fully or partially connected	Has to be mandatorily completely connected
Processing nodes in different layers shares a common neural model	Hidden nodes operate very differently and have a different purpose
Argument of hidden function activation function is the inner product of the inputs and the weights	The argument of each hidden unit activation function is the distance between the input and the weights
Trained with a single global supervised algorithm	RBF networks are usually trained one later at a time
Training is slower compared to RBF	Training is comparitely faster than MLP
After training MLP is much faster than RBF	After training RBF is much slower than MLP

Perceptron in Machine Learning

In Machine Learning and Artificial Intelligence, Perceptron is the most commonly used term for all folks. It is the primary step to learn Machine Learning and Deep Learning technologies, which consists of a set of weights, input values or scores, and a threshold. ***Perceptron is a building block of an Artificial Neural Network***. Initially, in the mid of 19th century, **Mr. Frank Rosenblatt** invented the Perceptron for performing certain calculations to detect input data capabilities or business intelligence. Perceptron is a linear Machine Learning algorithm used for supervised learning for various binary classifiers. This algorithm enables neurons to learn elements and processes them one by one during preparation. In this tutorial, "Perceptron in Machine Learning," we will discuss in-depth knowledge of Perceptron and its basic functions in brief. Let's start with the basic introduction of Perceptron.

What is the Perceptron model in Machine Learning?

Perceptron is Machine Learning algorithm for supervised learning of various binary classification tasks. Further, ***Perceptron is also understood as an Artificial Neuron or neural network unit that helps to detect certain input data computations in business intelligence***.

Perceptron model is also treated as one of the best and simplest types of Artificial Neural networks. However, it is a supervised learning algorithm of binary classifiers. Hence, we can consider it as a single-layer neural network with four main parameters, i.e., **input values, weights and Bias, net sum, and an activation function.**

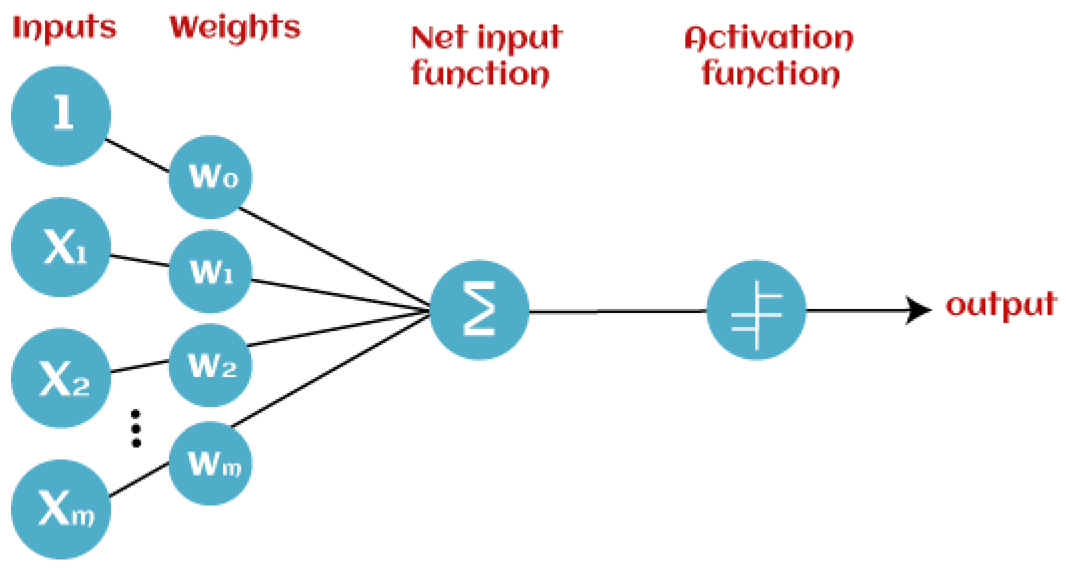
What is Binary classifier in Machine Learning?

In Machine Learning, binary classifiers are defined as the function that helps in deciding whether input data can be represented as vectors of numbers and belongs to some specific class.

Binary classifiers can be considered as linear classifiers. In simple words, we can understand it as a ***classification algorithm that can predict linear predictor function in terms of weight and feature vectors.***

Basic Components of Perceptron

Mr. Frank Rosenblatt invented the perceptron model as a binary classifier which contains three main components. These are as follows:



- **Input Nodes or Input Layer:**

This is the primary component of Perceptron which accepts the initial data into the system for further processing. Each input node contains a real numerical value.

- **Wight and Bias:**

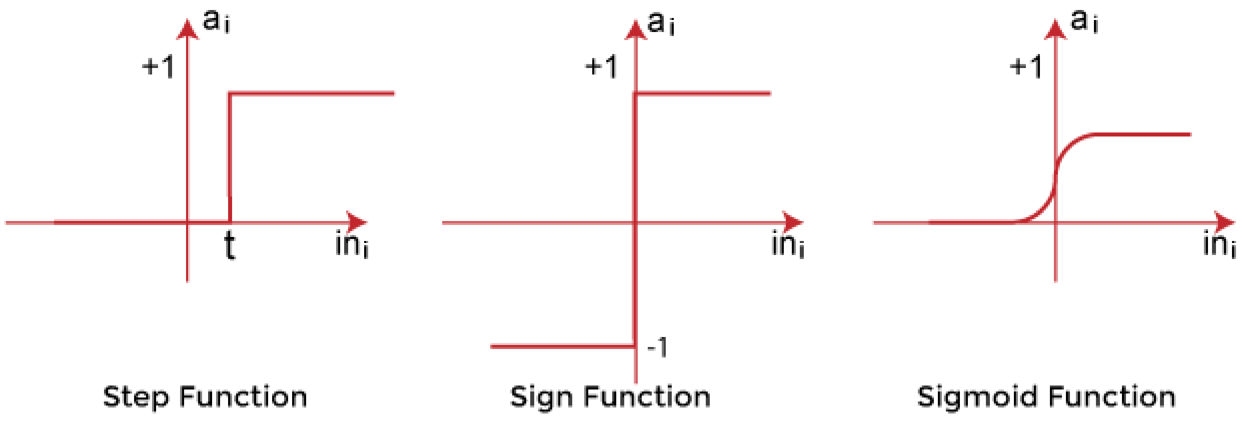
Weight parameter represents the strength of the connection between units. This is another most important parameter of Perceptron components. Weight is directly proportional to the strength of the associated input neuron in deciding the output. Further, Bias can be considered as the line of intercept in a linear equation.

- **Activation Function:**

These are the final and important components that help to determine whether the neuron will fire or not. Activation Function can be considered primarily as a step function.

Types of Activation functions:

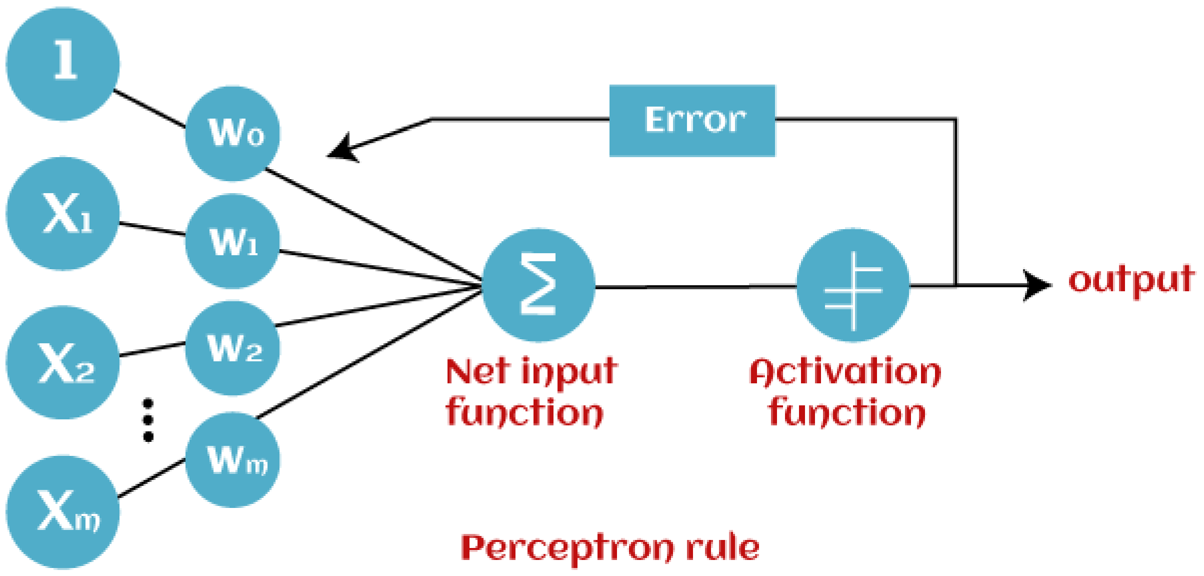
- Sign function
- Step function, and
- Sigmoid function



The data scientist uses the activation function to take a subjective decision based on various problem statements and forms the desired outputs. Activation function may differ (e.g., Sign, Step, and Sigmoid) in perceptron models by checking whether the learning process is slow or has vanishing or exploding gradients.

How does Perceptron work?

In Machine Learning, Perceptron is considered as a single-layer neural network that consists of four main parameters named input values (Input nodes), weights and Bias, net sum, and an activation function. The perceptron model begins with the multiplication of all input values and their weights, then adds these values together to create the weighted sum. Then this weighted sum is applied to the activation function 'f' to obtain the desired output. This activation function is also known as the **step function** and is represented by **'f'**.



This step function or Activation function plays a vital role in ensuring that output is mapped between required values (0,1) or (-1,1). It is important to note that the weight of input is indicative of the strength of a node. Similarly, an input's bias value gives the ability to shift the activation function curve up or down.

Perceptron model works in two important steps as follows:

Step-1

In the first step first, multiply all input values with corresponding weight values and then add them to determine the weighted sum. Mathematically, we can calculate the weighted sum as follows:

∑wi*xi = x1*w1 + x2*w2 +…wn*xn

Add a special term called **bias 'b'** to this weighted sum to improve the model's performance.

∑wi*xi + b

Step-2

In the second step, an activation function is applied with the above-mentioned weighted sum, which gives us output either in binary form or a continuous value as follows:

Y = f(∑wi*xi + b)

Types of Perceptron Models

Based on the layers, Perceptron models are divided into two types. These are as follows:

1. Single-layer Perceptron Model
2. Multi-layer Perceptron model

Single Layer Perceptron Model:

This is one of the easiest Artificial neural networks (ANN) types. A single-layered perceptron model consists feed-forward network and also includes a threshold transfer function inside the model. The main objective of the single-layer perceptron model is to analyze the linearly separable objects with binary outcomes.

In a single layer perceptron model, its algorithms do not contain recorded data, so it begins with inconstantly allocated input for weight parameters. Further, it sums up all inputs (weight). After adding all inputs, if the total sum of all inputs is more than a pre-determined value, the model gets activated and shows the output value as +1.

If the outcome is same as pre-determined or threshold value, then the performance of this model is stated as satisfied, and weight demand does not change. However, this model consists of a few discrepancies triggered when multiple weight inputs values are fed into the model. Hence, to find desired output and minimize errors, some changes should be necessary for the weights input.

"*Single-layer perceptron can learn only linearly separable patterns.*"

Multi-Layered Perceptron Model:

Like a single-layer perceptron model, a multi-layer perceptron model also has the same model structure but has a greater number of hidden layers.

The multi-layer perceptron model is also known as the Backpropagation algorithm, which executes in two stages as follows:

- **Forward Stage:** Activation functions start from the input layer in the forward stage and terminate on the output layer.
- **Backward Stage:** In the backward stage, weight and bias values are modified as per the model's requirement. In this stage, the error between actual output and demanded originated backward on the output layer and ended on the input layer.

Hence, a multi-layered perceptron model has considered as multiple artificial neural networks having various layers in which activation function does not remain linear, similar to a single layer perceptron model. Instead of linear, activation function can be executed as sigmoid, TanH, ReLU, etc., for deployment.

A multi-layer perceptron model has greater processing power and can process linear and non-linear patterns. Further, it can also implement logic gates such as AND, OR, XOR, NAND, NOT, XNOR, NOR.

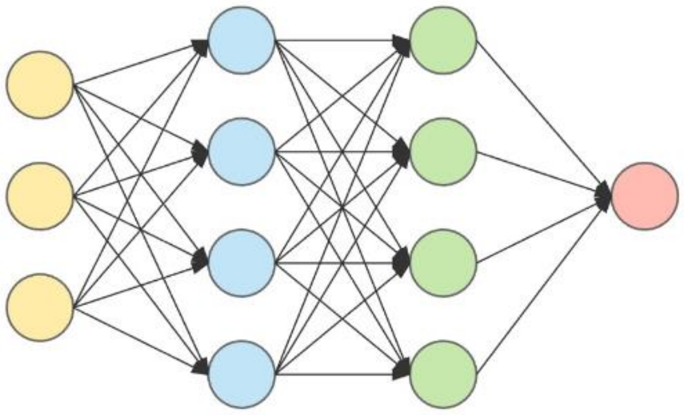


Fig – MLP

In the Neural Network Model, input data (yellow) are processed against a hidden layer (blue) and modified against more hidden layers (green) to produce the final output (red).

The First Layer:The 3 yellow perceptrons are making 3 simple decisions based on the input evidence. Each single decision is sent to the 4 perceptrons in the next layer.

The Second Layer:The blue perceptrons are making decisions by weighing the results from the first layer. This layer make more complex decisions at a more abstract level than the first layer.

The Third Layer:Even more complex decisions are made by the green perceptons.

Advantages of Multi-Layer Perceptron:

- A multi-layered perceptron model can be used to solve complex non-linear problems.
- It works well with both small and large input data.
- It helps us to obtain quick predictions after the training.
- It helps to obtain the same accuracy ratio with large as well as small data.

Disadvantages of Multi-Layer Perceptron:

- In Multi-layer perceptron, computations are difficult and time-consuming.
- In multi-layer Perceptron, it is difficult to predict how much the dependent variable affects each independent variable.
- The model functioning depends on the quality of the training.

Perceptron Function

Perceptron function ''f(x)'' can be achieved as output by multiplying the input 'x' with the learned weight coefficient 'w'.

Mathematically, we can express it as follows:

f(x)=1; if w.x+b>0

otherwise, f(x)=0

- 'w' represents real-valued weights vector
- 'b' represents the bias
- 'x' represents a vector of input x values.

Characteristics of Perceptron

The perceptron model has the following characteristics.

1. Perceptron is a machine learning algorithm for supervised learning of binary classifiers.
2. In Perceptron, the weight coefficient is automatically learned.
3. Initially, weights are multiplied with input features, and the decision is made whether the neuron is fired or not.
4. The activation function applies a step rule to check whether the weight function is greater than zero.
5. The linear decision boundary is drawn, enabling the distinction between the two linearly separable classes +1 and -1.
6. If the added sum of all input values is more than the threshold value, it must have an output signal; otherwise, no output will be shown.

Limitations of Perceptron Model

A perceptron model has limitations as follows:

- The output of a perceptron can only be a binary number (0 or 1) due to the hard limit transfer function.

- Perceptron can only be used to classify the linearly separable sets of input vectors. If input vectors are non-linear, it is not easy to classify them properly.

Future

The future of the Perceptron model is much bright and significant as it helps to interpret data by building intuitive patterns and applying them in the future. Machine learning is a rapidly growing technology of Artificial Intelligence that is continuously evolving and in the developing phase; hence the future of perceptron technology will continue to support and facilitate analytical behavior in machines that will, in turn, add to the efficiency of computers.

The perceptron model is continuously becoming more advanced and working efficiently on complex problems with the help of artificial neurons.

Perceptron Example

Imagine a perceptron (in your brain).

The perceptron tries to decide if you should go to a concert.

Is the artist good? Is the weather good?

What weights should these facts have?

Criteria	**Input**	**Weight**
Artists is Good	**x1** = 0 or 1	**w1** = 0.7
Weather is Good	**x2** = 0 or 1	**w2** = 0.6
Friend will Come	**x3** = 0 or 1	**w3** = 0.5

Food is Served	**x4** = 0 or 1	**w4** = 0.3
Location is Good	**x5** = 0 or 1	**w5** = 0.4

The Perceptron Algorithm

Frank Rosenblatt suggested this algorithm:

1. Set a threshold value
2. Multiply all inputs with its weights
3. Sum all the results
4. Activate the output

1. Set a threshold value:

- Threshold = 1.5

2. Multiply all inputs with its weights:

- x1 * w1 = 1 * 0.7 = 0.7
- x2 * w2 = 0 * 0.6 = 0
- x3 * w3 = 1 * 0.5 = 0.5
- x4 * w4 = 0 * 0.3 = 0
- x5 * w5 = 1 * 0.4 = 0.4

3. Sum all the results:

- 0.7 + 0 + 0.5 + 0 + 0.4 = 1.6 (The Weighted Sum)

4. Activate the Output:

- Return true if the sum > 1.5 ("Yes I will go to the Concert")

Note

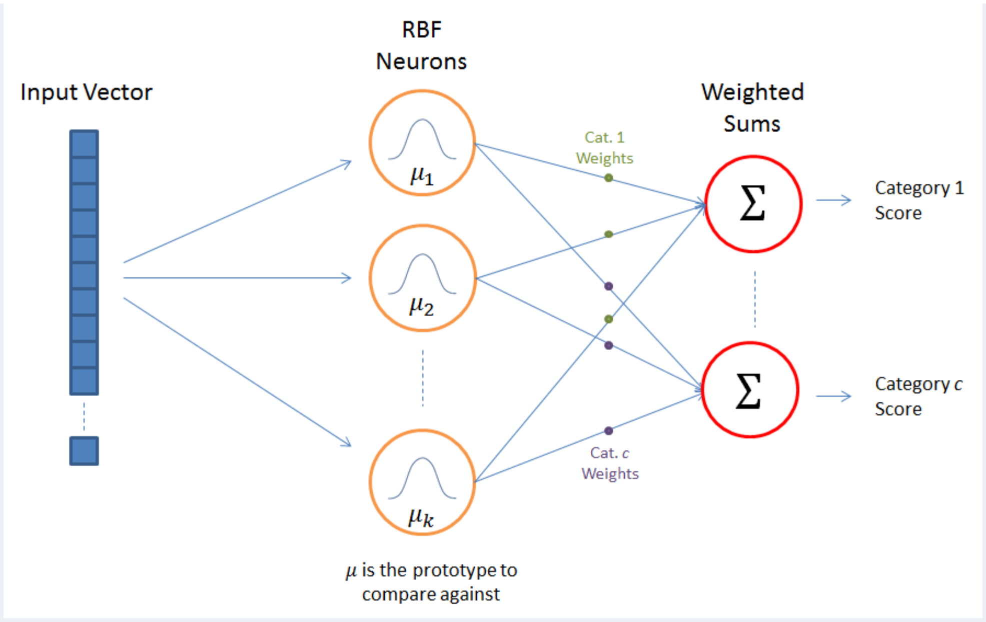
If the weather weight is 0.6 for you, it might different for someone else. A higher weight means that the weather is more important to them.

If the threshold value is 1.5 for you, it might be different for someone else. A lower threshold means they are more wanting to go to the concert.

Theory

- A **Radial Basis Function Network (RBFN)** is a particular type of neural network. The RBFN approach is more intuitive than MLP. An RBFN performs classification by measuring the input’s similarity to examples from the training set. Each RBFN neuron stores a “prototype”, which is just one of the examples from the training set. When we want to classify a new input, each neuron computes the Euclidean distance between the input and its prototype. Thus, if the input more closely resembles the class A prototypes than the class B prototypes, it is classified as class A.

  RBF Network Architecture



The above illustration shows the typical architecture of an RBF Network. It consists of an input vector, a layer of RBF neurons, and an output layer with one node per category or class of data.

The Input Vector
The input vector is the n-dimensional vector that you are trying to classify. The entire input vector is shown to each of the RBF neurons.

The RBF Neurons:
Each RBF neuron stores a “prototype” vector which is just one of the vectors from the training set.

- Each RBF neuron compares the input vector to its prototype and outputs a value between 0 and 1 which is a measure of similarity.
- If the input is equal to the prototype, then the output of that RBF neuron will be 1. As the distance between the input and prototype grows, the response falls off exponentially towards 0.
- The shape of the RBF neuron’s response is a bell curve, as illustrated in the network architecture diagram. The neuron’s response value is also called its “activation” value.
- The prototype vector is also often called the neuron’s “centre”, since it’s the value at the centre of the bell curve.

The Output Nodes:

The output of the network consists of a set of nodes, one per category that we are trying to classify.

- Each output node computes a sort of score for the associated category. Typically, a classification decision is made by assigning the input to the category with the highest score.
- The score is computed by taking a weighted sum of the activation values from every RBF neuron.
- By weighted sum, we mean that an output node associates a weight value with each of the RBF neurons, and multiplies the neuron's activation by this weight before adding it to the total response.
- Because each output node is computing the score for a different category, every output node has its own set of weights. The output node will typically give positive weight to the RBF neurons that belong to its category, and a negative weight to the others.

RBF Neuron Activation Function:

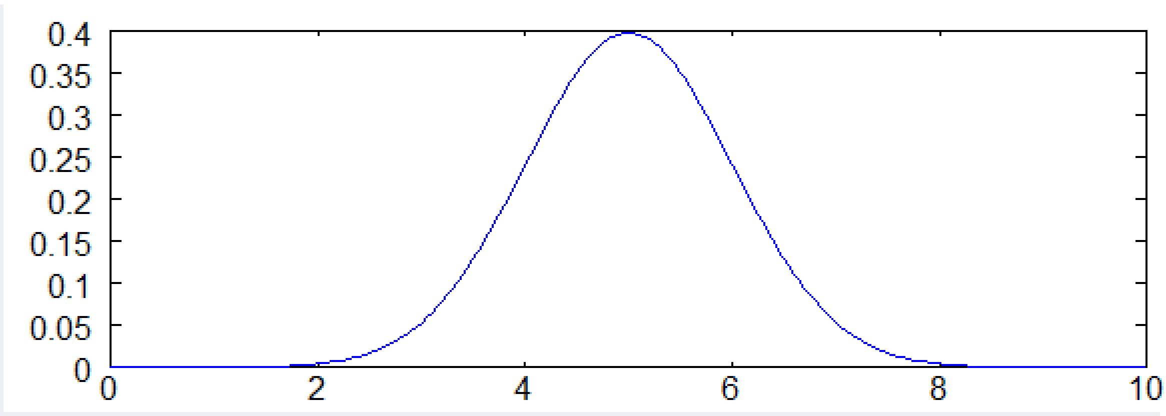
Each RBF neuron computes a measure of the similarity between the input and its prototype vector (taken from the training set).

- Input vectors which are more similar to the prototype return a result closer to 1.
- There are different possible choices of similarity functions, but the most popular is based on the Gaussian.
- Below is the equation for a Gaussian with a one-dimensional input.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Where x is the input, mu is the mean, and sigma is the standard deviation.

- This produces the familiar bell curve shown below, which is centered at the mean, mu (in the below plot the mean is 5 and sigma is 1).



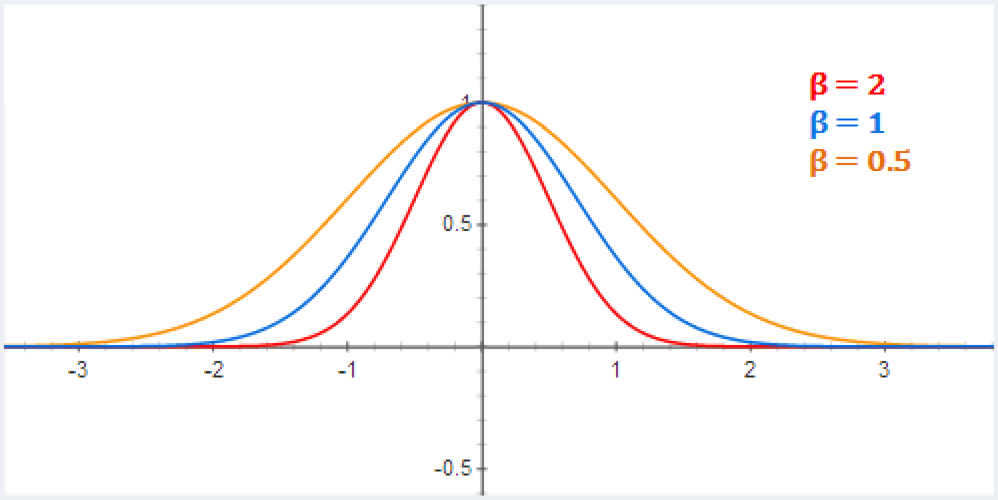
The RBF neuron activation function is slightly different, and is typically written as:

$$\varphi(x) = e^{-\beta\|x-\mu\|^2}$$

In the Gaussian distribution, mu refers to the mean of the distribution. Here, it is the prototype vector which is at the centre of the bell curve.

- For the activation function, phi, we aren’t directly interested in the value of the standard deviation, sigma, so we make a couple simplifying modifications.
- The first change is that we’ve removed the outer coefficient, 1 / (sigma * sqrt(2 * pi)). This term normally controls the height of the Gaussian. Here, though, it is redundant with the weights applied by the output nodes.
- During training, the output nodes will learn the correct coefficient or “weight” to apply to the neuron’s response. The second change is that we’ve replaced the inner coefficient, 1 / (2 * sigma^2), with a single parameter ‘beta’.
- This beta coefficient controls the width of the bell curve. Again, in this context, we don’t care about the value of sigma, we just care that there’s some coefficient which is controlling the width of the bell curve.

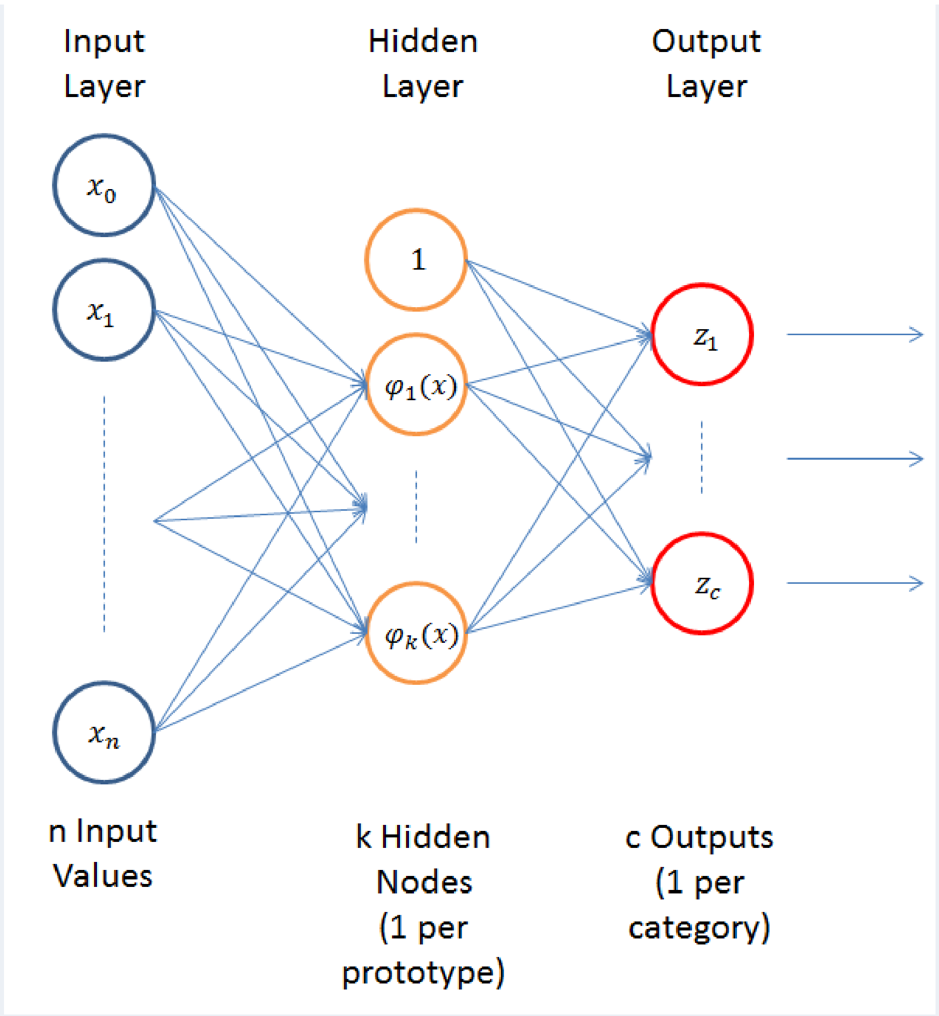
- So we simplify the equation by replacing the term with a single variable.



RBF Neuron activation for different values of beta There is also a slight change in notation here when we apply the equation to n-dimensional vectors. The double bar notation in the activation equation indicates that we are taking the Euclidean distance between x and mu, and squaring the result. For the 1-dimensional Gaussian, this simplifies to just (x - mu)^2. It’s important to note that the underlying metric here for evaluating the similarity between an input vector and a prototype is the Euclidean distance between the two vectors. Also, each RBF neuron will produce its largest response when the input is equal to the prototype vector. This allows to take it as a measure of similarity, and sum the results from all of the RBF neurons. As we move out from the prototype vector, the response falls off exponentially. Recall from the RBFN architecture illustration that the output node for each category takes the weighted sum of every RBF neuron in the network–in other words, every neuron in the network will have some influence over the classification decision. The exponential fall off of the activation function, however, means that the neurons whose prototypes are far from the input vector will actually contribute very little to the result.

RBFN as a Neural Network

Below is another version of the RBFN architecture diagram.



Here the RBFN is viewed as a "3-layer network" where the input vector is the first layer, the second "hidden" layer is the RBF neurons, and the third layer is the output layer containing linear combination neurons.

One bit of terminology that really had people confused for a while is that the prototype vectors used by the RBFN neurons are sometimes referred to as the "input weights".

It is generally think of weights as being coefficients, meaning that the weights will be multiplied against an input value. Here, though, we're computing the distance between the input vector and the "input weights" (the prototype vector).

Rule Base Classifier

- Rule-based classifiers are just another type of classifier which makes the class decision depending by using various "if..else" rules.
- These rules are easily interpretable and thus these classifiers are generally used to generate descriptive models.
- The condition used with "if" is called the **antecedent** and the predicted class of each rule is called the **consequent**.

IF-THEN Rules

Rule-based classifier makes use of a set of IF-THEN rules for classification. We can express a rule in the following from −

IF condition THEN conclusion

Let us consider a rule R1,

```
R1: IF age = youth AND student = yes
  THEN buy_computer = yes
```

Points to remember −

- The IF part of the rule is called **rule antecedent** or **precondition**.
- The THEN part of the rule is called **rule consequent**.
- The antecedent part the condition consist of one or more attribute tests and these tests are logically ANDed.
- The consequent part consists of class prediction.

Note − We can also write rule R1 as follows −

```
R1: (age = youth) ^ (student = yes))(buys computer = yes)
```

If the condition holds true for a given tuple, then the antecedent is satisfied.

Rule Extraction

Here we will learn how to build a rule-based classifier by extracting IF-THEN rules from a decision tree.

Points to remember −

To extract a rule from a decision tree −

- One rule is created for each path from the root to the leaf node.
- To form a rule antecedent, each splitting criterion is logically ANDed.
- The leaf node holds the class prediction, forming the rule consequent.

Rule Induction Using Sequential Covering Algorithm

Sequential Covering Algorithm can be used to extract IF-THEN rules form the training data. We do not require to generate a decision tree first. In this algorithm, each rule for a given class covers many of the tuples of that class.

Some of the sequential Covering Algorithms are AQ, CN2, and RIPPER. As per the general strategy the rules are learned one at a time. For each time rules are learned, a tuple covered by the rule is removed and the process continues for the rest of the tuples. This is because the path to each leaf in a decision tree corresponds to a rule.

Note − The Decision tree induction can be considered as learning a set of rules simultaneously.

The Following is the sequential learning Algorithm where rules are learned for one class at a time. When learning a rule from a class Ci, we want the rule to cover all the tuples from class C only and no tuple form any other class.

```
Algorithm: Sequential Covering

Input:
D, a data set class-labeled tuples,
Att_vals, the set of all attributes and their possible values.

Output:  A Set of IF-THEN rules.
Method:
Rule_set={ }; // initial set of rules learned is empty

for each class c do

  repeat
    Rule = Learn_One_Rule(D, Att_valls, c);
    remove tuples covered by Rule form D;
  until termination condition;

  Rule_set=Rule_set+Rule; // add a new rule to rule-set
end for
return Rule_Set;
```

Note : Tuple is one record or one row.

Properties of rule-based classifiers:

- **Coverage:** The percentage of records which satisfy the antecedent conditions of a particular rule.

- The rules generated by the rule-based classifiers are generally not mutually exclusive, i.e. many rules can cover the same record.
- The rules generated by the rule-based classifiers may not be exhaustive, i.e. there may be some records which are not covered by any of the rules.
- The decision boundaries created by them is linear, but these can be much more complex than the decision tree because the many rules are triggered for the same record.

An obvious question, which comes into the mind after knowing that the rules are not mutually exclusive is that how would the class be decided in case different rules with different consequent cover the record.

There are two solutions to the above problem:

- Either rules can be **ordered**, i.e. the class corresponding to the highest priority rule triggered is taken as the final class.
- Otherwise, we can assign **votes** for each class depending on some their weights, i.e. the rules remain unordered.

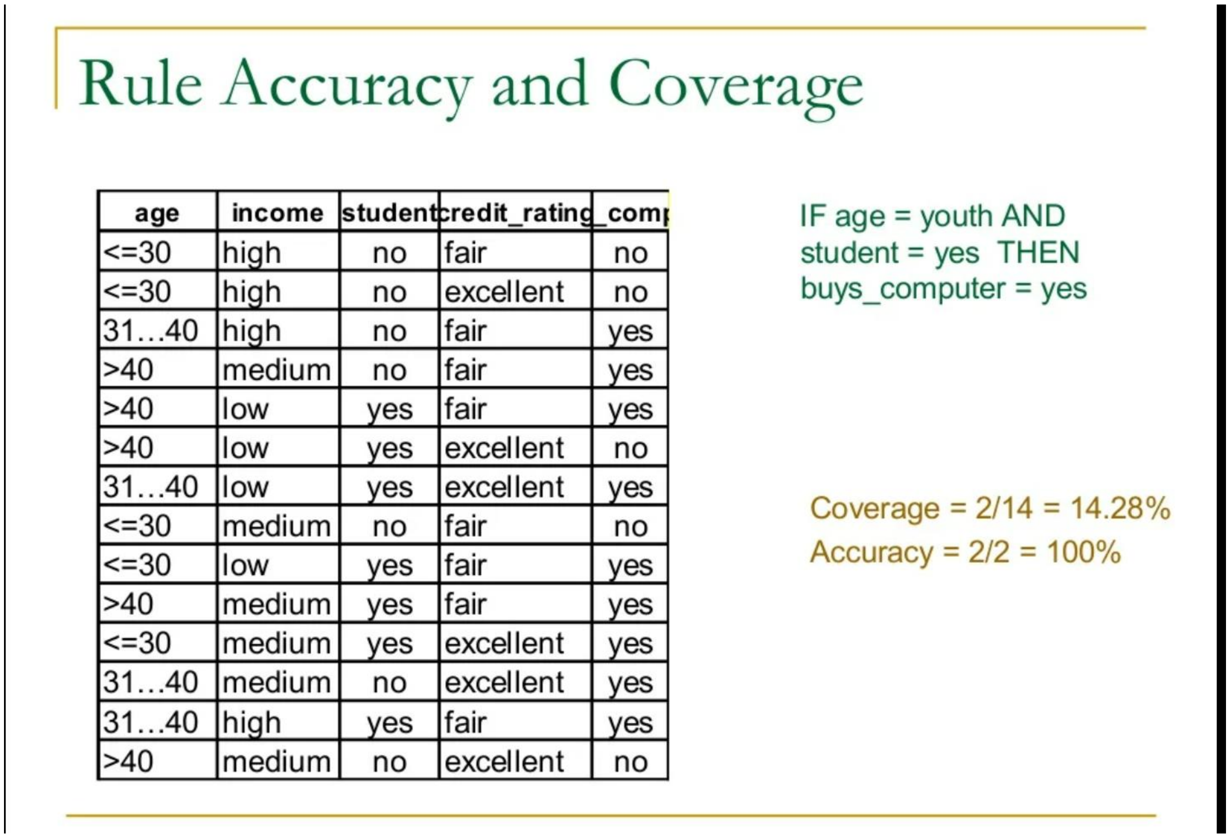
Rule-Based Classifier classify records by using a collection of “if…then…” rules.

(Condition)→Class Label

- The Left Hand Side is **rule antecedent** or **condition**
- The Right Hand Side is **rule consequent**
- **Coverage** of a rule - Fraction of records that satisfy the antecedent of a rule
- **Accuracy** of a rule - Fraction of records that satisfy both the antecedent and consequent of a rule

Rule-Based Classification

- Model – Rules
- Set of IF-THEN rules
  - IF *age* = youth AND *student* = yes THEN *buys_computer* = yes
  - Rule antecedent/precondition vs. rule consequent
- Assessment of a rule: *coverage* and *accuracy*
  - n_{covers} = # of tuples covered by R
  - $n_{correct}$ = # of tuples correctly classified by R
  - coverage(R) = n_{covers}/|D| /* D: training data set */
  - accuracy(R) = $n_{correct}$/ n_{covers}



Rule Accuracy and Coverage

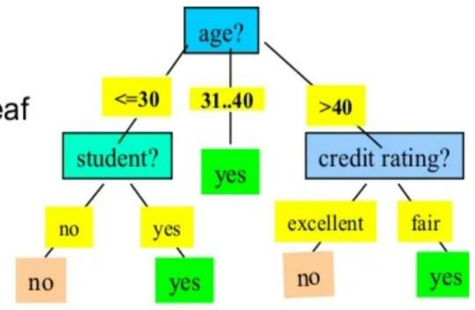
age	income	student	credit_rating	comp
<=30	high	no	fair	no
<=30	high	no	excellent	no
31…40	high	no	fair	yes
>40	medium	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
31…40	low	yes	excellent	yes
<=30	medium	no	fair	no
<=30	low	yes	fair	yes
>40	medium	yes	fair	yes
<=30	medium	yes	excellent	yes
31…40	medium	no	excellent	yes
31…40	high	yes	fair	yes
>40	medium	no	excellent	no

IF age = youth AND
student = yes THEN
buys_computer = yes

Coverage = 2/14 = 14.28%
Accuracy = 2/2 = 100%

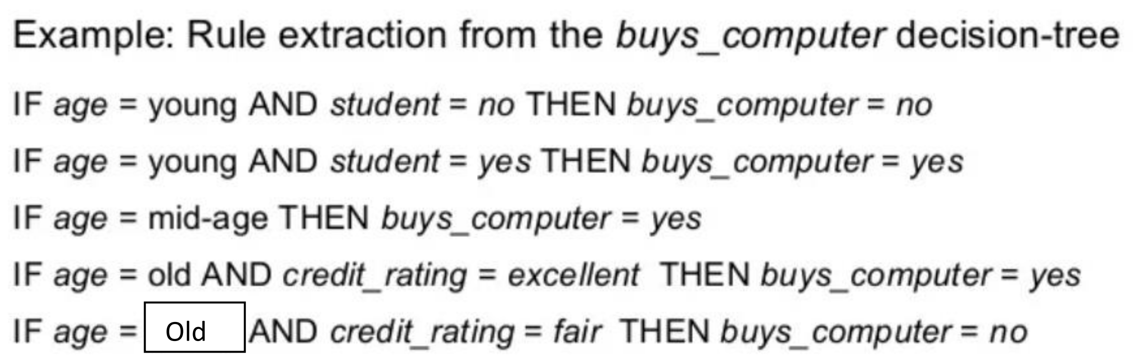
Write the set of rules for the following diagram.

- Rules are easier to understand than large trees
- One rule is created for each path from the root to a leaf
- Each attribute-value pair along a path forms a conjunction: the leaf holds the class prediction
- Rules are **mutually exclusive** and **exhaustive** (so unordered)



Consider

- <=30 as young age
- 31.40 as Mid-age
- >40 as old age



Example: Rule extraction from the *buys_computer* decision-tree

IF *age* = young AND *student* = *no* THEN *buys_computer* = *no*

IF *age* = young AND *student* = *yes* THEN *buys_computer* = *yes*

IF *age* = mid-age THEN *buys_computer* = *yes*

IF *age* = old AND *credit_rating* = *excellent* THEN *buys_computer* = *yes*

IF *age* = Old AND *credit_rating* = *fair* THEN *buys_computer* = *no*

Advantages of Rule-Based Classifiers

- As highly expressive as decision trees
- Easy to interpret
- Easy to generate
- Can classify new instances rapidly
- Performance comparable to decision trees
