

CLASSIFICATION

UNIT-III

Basic concepts:

What is Classification?

Classification is a **supervised learning** technique used to **categorize data** into **predefined classes** or **labels**.

(or)

In other words, Classification is a data mining technique that **categorizes items** in a collection based on some **predefined properties**.

Examples:

1. A **bank loans officer** needs analysis of her data to learn which loan applicants are “**safe**” and which are “**risky**” for the bank.
2. A **marketing manager** needs data analysis to help guess whether a customer with a given profile will **buy** a new computer.
3. A **medical researcher** wants to analyse breast cancer data to predict which one of three specific treatments a patient should receive.

In each of these examples, the data analysis task is **classification**, where a model or **classifier** is constructed to **predict class labels**, such as

- “**safe**” or “**risky**” for the loan application data;
- “**yes**” or “**no**” for the marketing data;
- “**treatment A**,” “**treatment B**,” or “**treatment C**” for the medical data.

These categories/class labels can be represented by **discrete values**, where the ordering among values has **no meaning**.

How does classification work?

Data classification is a two-step process, consisting of a **learning step** (where a classification model is constructed) and a **classification step** (where the model is used to predict class labels for given data).

- **Learning step (or training phase):**

In this, a classification algorithm builds the **classifier/model** by analyzing or “learning from” a **training dataset** and their associated class labels. and the learned/trained model or classifier is represented in the form of **classification rules**.

- **Classification step**

In this, **Test data** are used to estimate the accuracy of the classification rules. If the accuracy is considered acceptable, the rules can be applied to the classification of new data tuples. i.e., the classifier/model is used to predict class labels for the given new data.

Example:

A **bank loans officer** needs analysis of her data to learn which loan applicants are “**safe**” and which are “**risky**” for the bank.

- **Learning step (or training phase):**

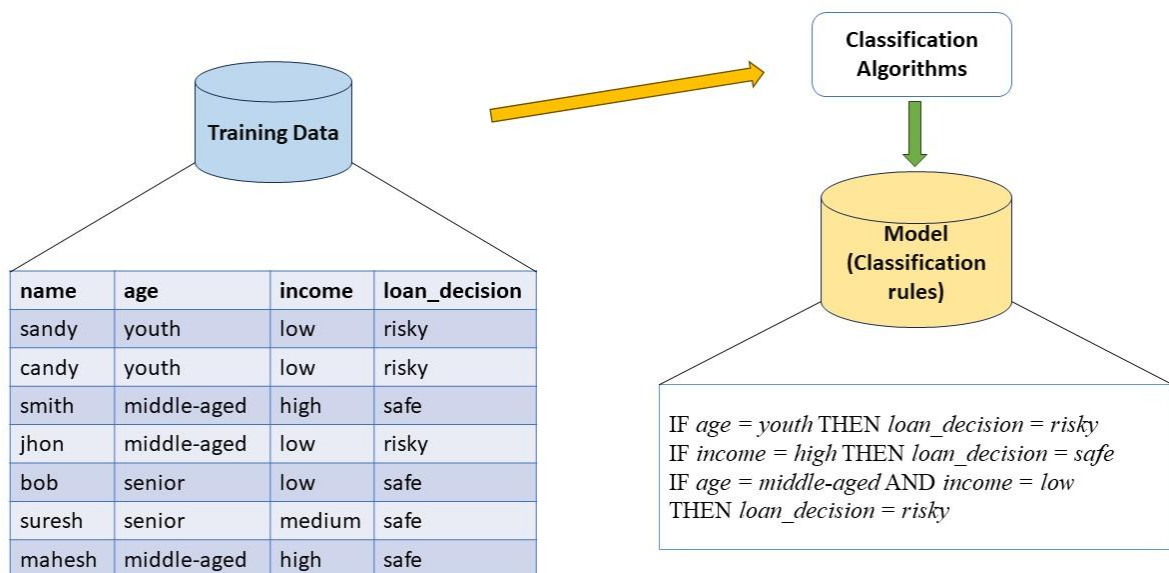


Fig: Learning step

- **Classification step (Using the Model in Prediction)**

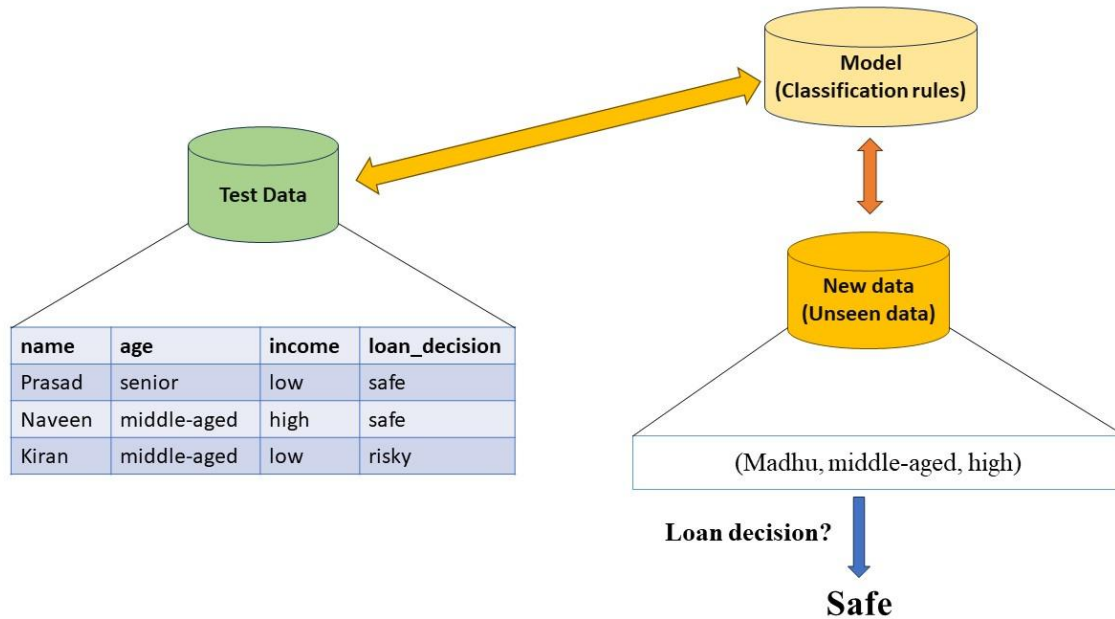


Fig: Classification step

What about classification accuracy?

The model/classifier is used for classification. First, the predictive accuracy of the classifier/model is estimated. If we were to use the **training data set** to measure the classifier's accuracy, this estimate would likely be **optimistic**, because the classifier tends to **overfit** the data (i.e., during learning it may incorporate some anomalies of the training data that are not present in the general data set overall). Therefore, a **test data set** is used, made up of test tuples and their associated class labels.

The **accuracy** of a classifier on a given **test data set** is the percentage of test set tuples that are correctly classified by the classifier. The associated class label of each test tuple is compared with the learned classifier's class prediction for that tuple.

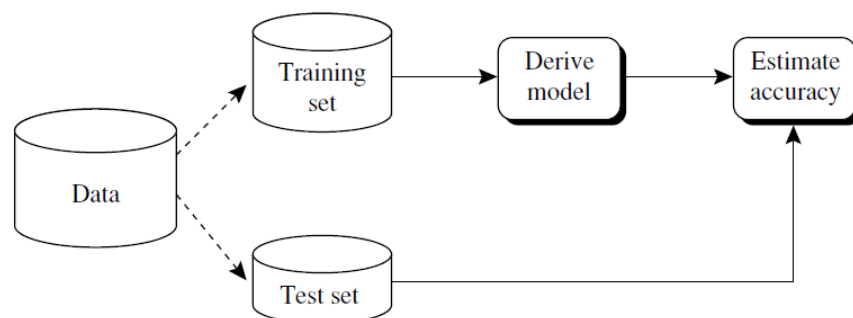


Fig: Estimate accuracy

Decision Tree Induction:

Decision Tree:

“What is decision tree?” A **decision tree** is a flowchart-like tree structure, where each *internal node* (nonleafnode) denotes a **test** on an attribute, each *branch* represents an *outcome* of the test, and each *leaf node* (or terminal node) holds a **class label**. The topmost node in a tree is the *root node*. It represents the best attribute selected for classification.

Decision Tree is a type of data mining technique that is used to build *Classification Models*. It builds classification models in the form of a tree-like structure, just like its name. This type of mining belongs to *supervised class learning*.

In *supervised learning*, the target result is already known. Decision trees can be used for both categorical and numerical data. The categorical data represent gender, marital status, etc. while the numerical data represent age, temperature, etc.

Example:

The following is a typical decision tree. It represents the concept *buys_computer*, that is, it predicts whether a customer at *AllElectronics* is likely to purchase a computer. Internal nodes are denoted by rectangles, and leaf nodes are denoted by ovals.

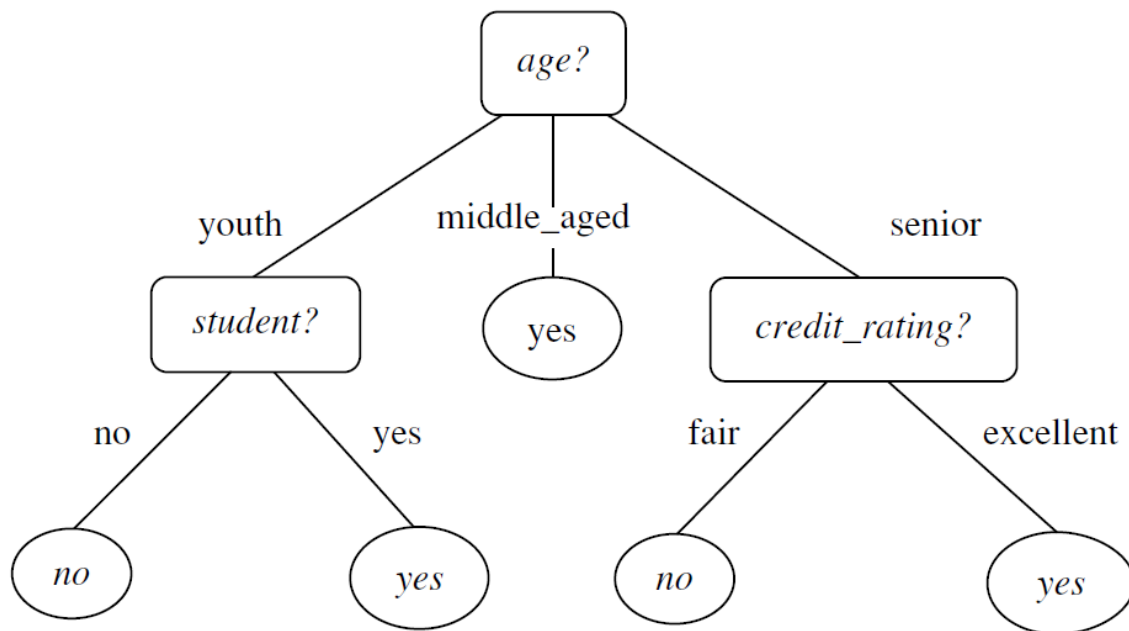


Fig: Decision tree

A decision tree for the concept *buys_computer*, indicating whether an *AllElectronics* customer is likely to purchase a computer. Each *internal* (nonleaf) node represents a **test** on an attribute. Each leaf node represents a **class** (either *buys_computer* = yes or *buys_computer* = no).

“How are decision trees used for classification?” Given a tuple, *X*, for which the associated class label is unknown, the attribute values of the tuple are tested against the decision tree. A path is traced from the root to a leaf node, which holds the class prediction for that tuple. Decision trees can easily be converted to classification rules.

Example, According to the decision tree provided above, the tuple (*age* = senior, *credit_rating* = excellent) is associated to the class label (*buys_computer* = yes).

“Why are decision tree classifiers so popular?” The construction of decision tree classifiers does not require any domain knowledge or parameter setting, and therefore is appropriate for exploratory knowledge discovery. Decision trees can handle multidimensional data.

Decision tree induction algorithms have been used for classification in many application areas such as medicine, manufacturing and production, financial analysis, astronomy, and molecular biology.

Decision Tree Induction:

In 1980, *J. Ross Quinlan*, a researcher in machine learning, created the **ID3** (Iterative Dichotomiser) algorithm, which is a *decision tree algorithm*. Later on, he introduced **C4.5**, which was the *successor of ID3*. Both ID3 and C4.5 use a *greedy approach*, where the trees are constructed in a top-down recursive divide-and-conquer manner without any backtracking.

In 1984, a group of statisticians published the book *Classification and Regression Trees* (**CART**), which described the generation of binary decision trees. *ID3* and *CART* were invented independently of one another at around the same time, yet follow a similar approach for learning decision trees from training tuples.

Decision Tree algorithm:

Generate_decision_tree. Generate a decision tree from the training tuples of data partition, D .

Input:

Data partition, D , which is a set of training tuples and their associated class labels;

attribute_list, the set of candidate attributes;

Attribute_selection_method, describes the method for selecting the best attribute for discrimination among tuples.

Output: A decision tree.

Method:

Step-1: Begin the tree with the root node, says R , which contains the complete dataset D .

Step-2: Find the best attribute in the dataset using Attribute Selection Method (ASM).

Step-3: Divide the R into subsets that contains possible values for the best attributes.

Step-4: Generate the decision tree node, which contains the best attribute.

Step-5: Recursively make new decision trees using the subsets of the dataset created in step -3. Continue this process until a stage is reached where you cannot further classify the nodes and called the final node as a leaf node.

Example: Suppose there is a person who has a job offer and wants to decide whether he should accept or decline the offer. Solve this problem, using the decision tree.

Input:

attribute_list: Salary, Distance and Cab_facility.

class_labels: Accept, Decline.

Method:

- The decision tree starts with the root node (Salary attribute).
- The root node splits further into the next decision node (Distance attribute) and one leaf node based on the corresponding labels.
- The next decision node further gets split into one decision node (Cab_facility attribute) and one leaf node.
- Finally, the decision node (Cab_facility attribute) splits into two leaf nodes (Accept and Decline).

Output:

The below diagram is the decision tree:

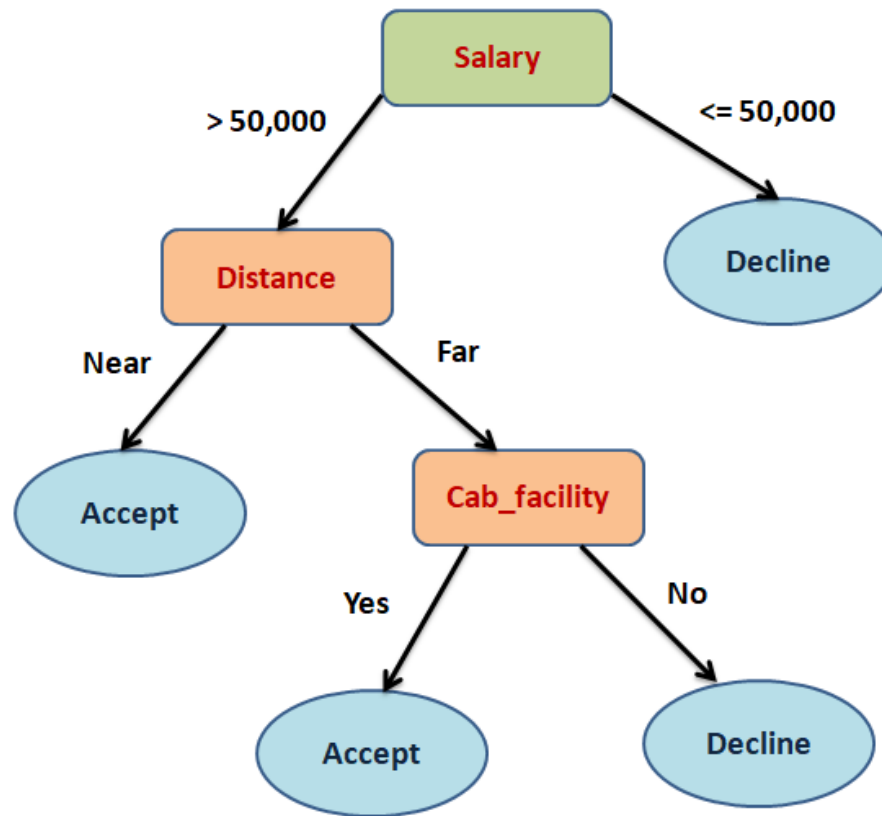


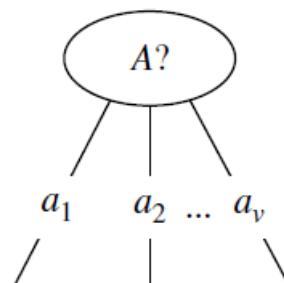
Fig: Decision tree

The algorithm calls *Attribute selection method* to determine the **splitting criterion**. The splitting criterion tells us which attribute to test at node N by determining the “best” way to separate or partition the tuples in D into individual classes.

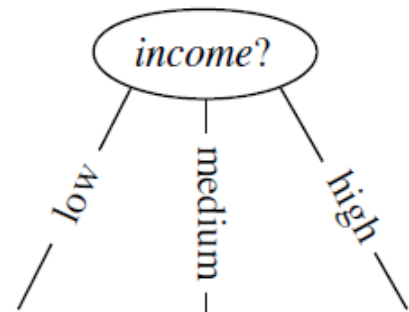
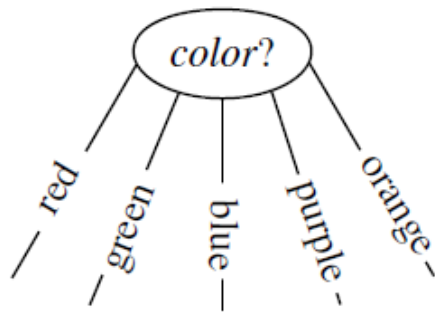
There are three possibilities for partitioning tuples based on the splitting criterion.

Let **A** be the splitting attribute,

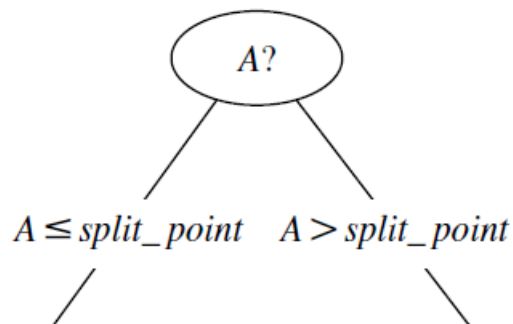
- (a) If **A** is *discrete-valued*, then one branch is grown for each known value of A.



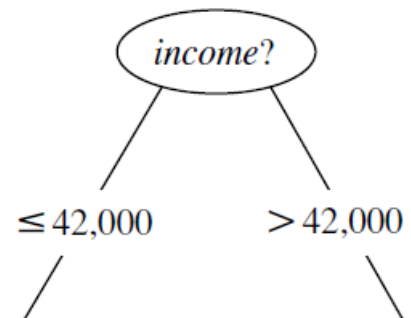
Example:



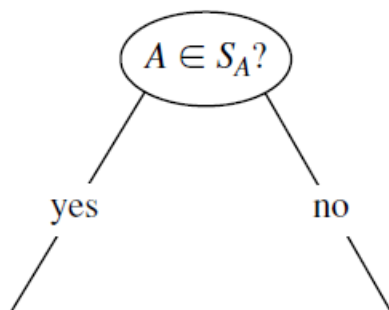
- (b) If **A** is *continuous-valued*, then two branches are grown, corresponding to $A \leq \text{split_point}$ and $A > \text{split_point}$.



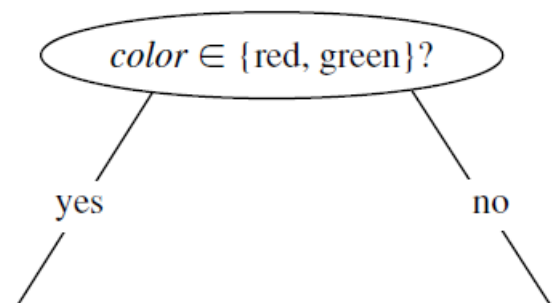
Example:



- (c) If **A** is *discrete-valued* and a *binary tree* must be produced, then the test is of the form $A \in S_A$, where S_A is the splitting subset for **A**.



Example:



Attribute Selection Measures:

When implementing a Decision tree, the primary challenge is to choose the most suitable attribute for the root node and sub-nodes. However, there is a technique known as *Attribute selection measure* that can help resolve such issues. With the help of this measurement, we can easily determine the best attribute for the nodes of the tree.

Here is a list of some attribute selection measures.

- **Entropy**

The calculation of entropy is performed for each attribute after every split, and the splitting process is then continued. A lower entropy indicates a better model, as it implies that the classes are split more effectively due to *reduced uncertainty*.

It is the measure of *impurity (or) uncertainty* in the data. It lies between 0 to 1 and is calculated using the below formula.

$$Entropy(D) = \sum_{i=1}^n -p_i * \log_2(p_i)$$

where p_i is the nonzero probability that an arbitrary tuple in D belongs to class C_i .

For example, let us see the formula for a use case with **2 classes**- *Yes* and *No*.

$$Entropy(\mathbf{D}) = -P(\text{Yes}) * \log_2(P(\text{Yes})) - P(\text{No}) * \log_2(P(\text{No}))$$

Here *Yes* and *No* are two classes and $\mathbf{D}$ is the total number of samples. $P(\text{Yes})$ is the probability of Yes and $P(\text{No})$ is the probability of No.

If there are **many classes** then the formula is

$$Entropy(\mathbf{D}) = -P_1 * \log_2(P_1) - P_2 * \log_2(P_2) - \dots - P_n * \log_2(P_n)$$

Here n is the number of classes.

- **Information gain**

This method is the main method that is used to *build decision trees*. It reduces the information that is required to classify the tuples. It reduces the number of tests that are needed to classify the given tuple.

- The attribute with the highest information gain is selected.
- Information gain is basically measuring the change in entropy.

Now, suppose we have to partition the tuples in D on some attribute A having v distinct values, $\{a_1, a_2, \dots, a_v\}$.

Then the expected information required to classify the tuple from D based on attribute A is

$$Entropy_A(D) = \sum_{j=1}^v \frac{|D_j|}{|D|} * Entropy(D_j)$$

The term $|D_j| / |D|$ acts as the weight of the j^{th} partition. $Entropy_A(D)$ is the expected information required to classify a tuple from D based on the partitioning by A .

Information gain is defined as the *difference* between the original information requirement and the new requirement (i.e., obtained after partitioning on A)

$$Gain(A) = Entropy(D) - Entropy_A(D)$$

The attribute A with the highest information gain, $Gain(A)$, is chosen as the splitting attribute at node N .

Example:

Let us illustrate the ID3 algorithm with a simple example of classifying whether customer buys a computer based on some conditions. (or) Construct a decision tree to classify “buys computer.”

Consider the following dataset:

age	income	student	credit_rating	buys_computer
youth	high	no	fair	no
youth	high	no	excellent	no
middle_aged	high	no	fair	yes
senior	medium	no	fair	yes
senior	low	yes	fair	yes
senior	low	yes	excellent	no
middle_aged	low	yes	excellent	yes
youth	medium	no	fair	no
youth	low	yes	fair	yes
senior	medium	yes	fair	yes
youth	medium	yes	excellent	yes
middle_aged	medium	no	excellent	yes
middle_aged	high	yes	fair	yes
senior	medium	no	excellent	no

Table-1: customer dataset

Solution:

In the given customer dataset, we have 5 attributes/columns, those are *age*, *income*, *student*, *credit_rating*, and *buys_computer*.

In which *age*, *income*, *student*, *credit_rating* attributes are categorized as **input features** and whereas *buys_computer* categorized as **output/target feature**.

Here, we are going to classify whether customer buys a computer or not based on the input features. And we will be going to use *ID3 algorithm* for building *decision tree*.

Let us start the solution:

First, we need to count the number of “yes” and “no” for output column / feature called “buys_computer.”, There are **5** “no” and **9** “yes”

So, we will calculate **entropy** for **output/target** variable.

The formula for **entropy** is

$$Entropy(D) = \sum_{i=1}^n -p_i * \log_2(p_i)$$

The given dataset with **2 classes**- *yes* and *no*. Then the formula of **Entropy** for **buys_computer** is

$$Entropy(\text{buys_computer}) = -P(\text{yes}) * \log_2(P(\text{yes})) - P(\text{no}) * \log_2(P(\text{no}))$$

Where,

- $P(\text{yes}) = p(\text{yes}) / (p(\text{yes}) + p(\text{no}))$
 $= 9 / (9 + 5)$
 $= 9/14$
for “**yes**” component of buys_computer.
- $P(\text{no}) = p(\text{no}) / (p(\text{yes}) + p(\text{no}))$
 $= 5 / (9 + 5)$
 $= 5 / 14$
for “**no**” component of buys_computer.

Let us substitute the values in the entropy formula,

$$\begin{aligned} Entropy(\text{buys_computer}) &= - (9/14) * \log_2(9/14) - (5/14) * \log_2(5/14) \\ &= - (0.64) * \log_2(0.64) - (0.36) * \log_2(0.36) \\ &= 0.412 + 0.53 \\ &= \mathbf{0.94\text{bits}} \end{aligned}$$

$Entropy(\text{buys_computer}) = \mathbf{0.94\text{bits}}$

Once we are ready with the Entropy of target variable, we will find out the Entropy for *each column/variable/attribute* with respect to *target variable* to find most homogeneous sample.

Let's extract the **age column** from given dataset.

		buys_computer		
		yes	no	
age	youth	2	3	5
	middle_aged	4	0	4
	senior	3	2	5
				14

So, here we can see that

- *youth* has 2 yes, 3 no
- *middle_aged* has 4 yes and 0 no
- *senior* has 3 yes and 2 no.

Now we can calculate the Entropy for each value of **age** column.

$$\begin{aligned} \text{Entropy}(\text{youth}) &= - (2/5) * \log_2(2/5) - (3/5) * \log_2(3/5) \\ &= \mathbf{0.97} \end{aligned}$$

$$\begin{aligned} \text{Entropy}(\text{senior}) &= - (3/5) * \log_2(3/5) - (2/5) * \log_2(2/5) \\ &= \mathbf{0.97} \end{aligned}$$

$$\begin{aligned} \text{Entropy}(\text{middle_aged}) &= - (4/4) * \log_2(4/4) - (0/4) * \log_2(0/4) \\ &= \mathbf{0.0} \end{aligned}$$

The Entropy of *middle_aged* is **0** i.e., *middle_aged* has only “yes” component.

Note: If the Entropy is Zero (0) that will be categorized as pure subset.

So, once we are ready with Entropy for each value in **age** column then we will find out the *Entropy* for complete **age** column with respect to **buys_computer** column.

$$\text{Entropy}(\text{buys_computer}, \text{age}) =$$

$$\begin{aligned} &P(\text{youth}) * E(\text{youth}) + P(\text{middle_aged}) * E(\text{middle_aged}) + P(\text{senior}) * E(\text{senior}) \\ &= (5/14) * 0.97 + (4/14) * 0 + (5/14) * 0.97 \\ &= 0.693\text{bits} \end{aligned}$$

$$\text{Entropy}(\text{buys_computer}, \text{age}) = \mathbf{0.693\text{bits}}$$

Now, we need to calculate the information gain of column age is

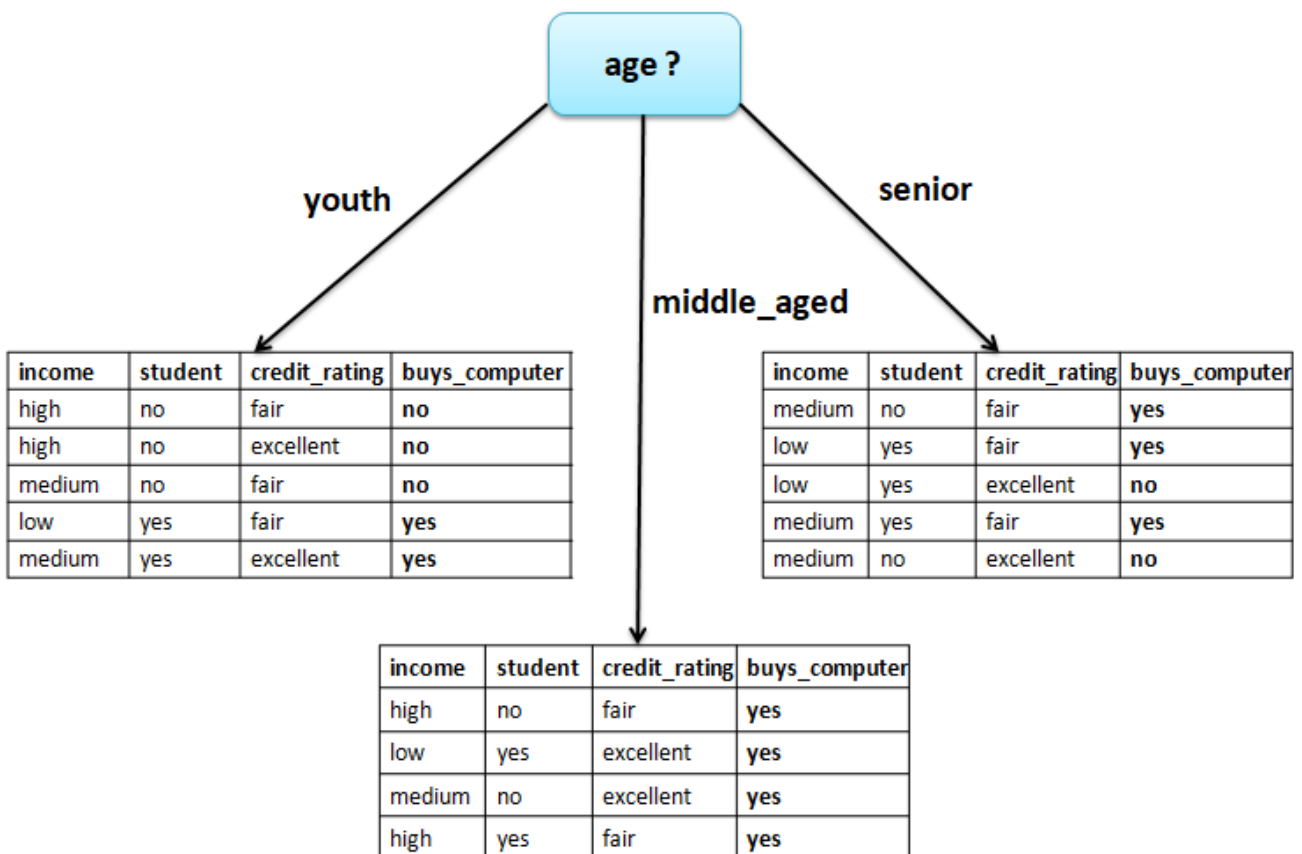
Information Gain (*buys_computer*, *age*) =

$$\begin{aligned} & \text{Entropy}(\text{buys_computer}) - \text{Entropy}(\text{buys_computer}, \text{age}) \\ &= 0.94 - 0.693 \\ &= \mathbf{0.247\text{bits}} \end{aligned}$$

Information Gain (*buys_computer*, *age*) = **0.247bits**

Similarly, we can compute *Information Gain* (*buys_computer*, *income*) is **0.029** bits, *Information Gain* (*buys_computer*, *student*) is **0.151** bits, and *Information Gain* (*buys_computer*, *credit_rating*) is **0.048** bits.

The attribute **age** has the *highest information gain* and therefore becomes the *splitting attribute* at the root node of the decision tree. Branches are grown for each outcome of **age**. The tuples are shown partitioned accordingly.



Notice that the tuples falling into the partition for *age* = *middle_aged* all belong to the same class. Because they all belong to class “yes,” a leaf should therefore be created at the end of this branch and labelled “yes.”

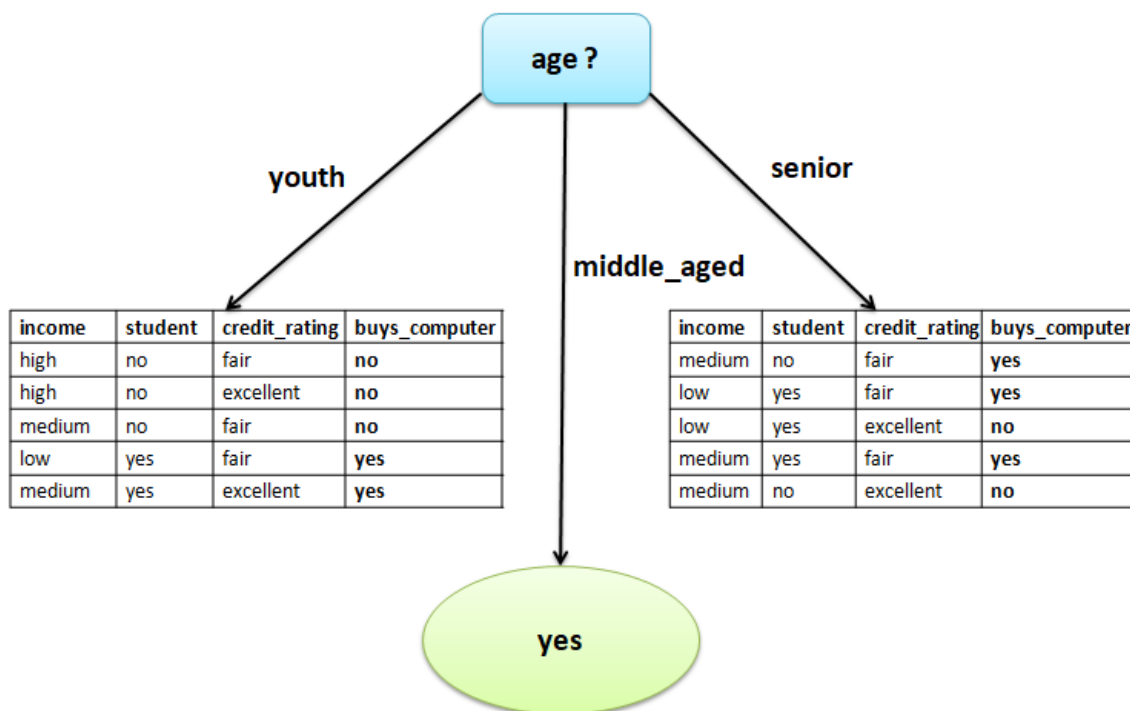


Fig: Decision tree for *age* attribute.

Let’s continue with the next level. If the *age* is “youth” then the dataset is

income	student	credit_rating	buys_computer
high	no	fair	no
high	no	excellent	no
medium	no	fair	no
low	yes	fair	yes
medium	yes	excellent	yes

In the above, *income*, *student* and *credit_rating* are categorized as input attributes and *buys_computer* is categorized as output attribute. There are 5 instances among those 2 yes and 3 no for *buys_computer* attribute.

Now, we will calculate the *Information Gain* for all the attributes with respect to target/class attribute called *buys_computer*. To calculate *Information Gain* first, we need to calculate the *Entropy* for all the attributes.

Let's calculate the **Entropy** for class/target attribute **buys_computer** is

$$\begin{aligned}\text{Entropy}(\mathbf{buys_computer}) &= -P(\text{yes}) \cdot \log_2(P(\text{yes})) - P(\text{no}) \cdot \log_2(P(\text{no})) \\ &= - (2/5) \cdot \log_2(2/5) - (3/5) \cdot \log_2(3/5) \\ &= \mathbf{0.97\text{bits}}\end{aligned}$$

$$\text{Entropy}(\mathbf{buys_computer}) = \mathbf{0.97\text{bits}}$$

Once we are ready with the Entropy of target variable, we will find out the Entropy for *each column/variable/attribute* with respect to *target variable* to find most homogeneous sample.

Let's extract the **income** attribute from given dataset.

		buys_computer		
		yes	no	
income	low	1	0	1
	medium	1	1	2
	high	0	2	2
				5

So, here we can see that

- *low* has 1 yes, 0 no
- *medium* has 1 yes and 1 no
- *high* has 0 yes and 2 no.

Now we can calculate the Entropy for each value of **income** column.

$$\begin{aligned}\text{Entropy}(\mathbf{low}) &= - (1/1) \cdot \log_2(1/1) - (0/1) \cdot \log_2(0/1) \\ &= \mathbf{0}\end{aligned}$$

$$\begin{aligned}\text{Entropy}(\mathbf{medium}) &= - (1/2) \cdot \log_2(1/2) - (1/2) \cdot \log_2(1/2) \\ &= \mathbf{1}\end{aligned}$$

$$\begin{aligned}\text{Entropy}(\mathbf{high}) &= - (0/2) \cdot \log_2(0/2) - (2/2) \cdot \log_2(2/2) \\ &= \mathbf{0}\end{aligned}$$

So, once we are ready with *Entropy* for each value in **income** column then we will find out the *Entropy* for complete **income** column with respect to **buys_computer** column.

Entropy (buys_computer, income) =

$$\begin{aligned} & P(\text{low}) * E(\text{low}) + P(\text{medium}) * E(\text{medium}) + P(\text{high}) * E(\text{high}) \\ &= (1/5) * 0 + (2/5) * 1 + (1/5) * 0 \\ &= 0.4\text{bits} \end{aligned}$$

$$\text{Entropy (buys_computer, income)} = \mathbf{0.4\text{bits}}$$

Now, we need to calculate the **information gain** of column **income** is

Information Gain (buys_computer, income) =

$$\begin{aligned} & \text{Entropy (buys_computer)} - \text{Entropy (buys_computer, income)} \\ &= 0.97 - 0.4 \\ &= \mathbf{0.57\text{bits}} \end{aligned}$$

$$\text{Information Gain (buys_computer, income)} = \mathbf{0.57\text{bits}}$$

Similarly, we can compute *Information Gain (buys_computer, student)* and *Information Gain (buys_computer, credit_rating)*.

Information Gain (buys_computer, student) =

$$\begin{aligned} & \text{Entropy (buys_computer)} - \text{Entropy (buys_computer, student)} \\ &= 0.97 - 0 \\ &= \mathbf{0.97\text{bits}} \end{aligned}$$

$$\text{Information Gain (buys_computer, student)} = \mathbf{0.97\text{bits}}$$

Information Gain (buys_computer, credit_rating) =

$$\begin{aligned} & \text{Entropy (buys_computer)} - \text{Entropy (buys_computer, credit_rating)} \\ &= 0.97 - 0.946 \\ &= \mathbf{0.024 \text{ bits}} \end{aligned}$$

$$\text{Information Gain (buys_computer, credit_rating)} = \mathbf{0.024\text{bits}}$$

The attribute **student** has the *highest information gain* and therefore becomes the *splitting attribute* at the next level node of the decision tree. Branches are grown for each outcome of **student**. The tuples are shown partitioned accordingly.

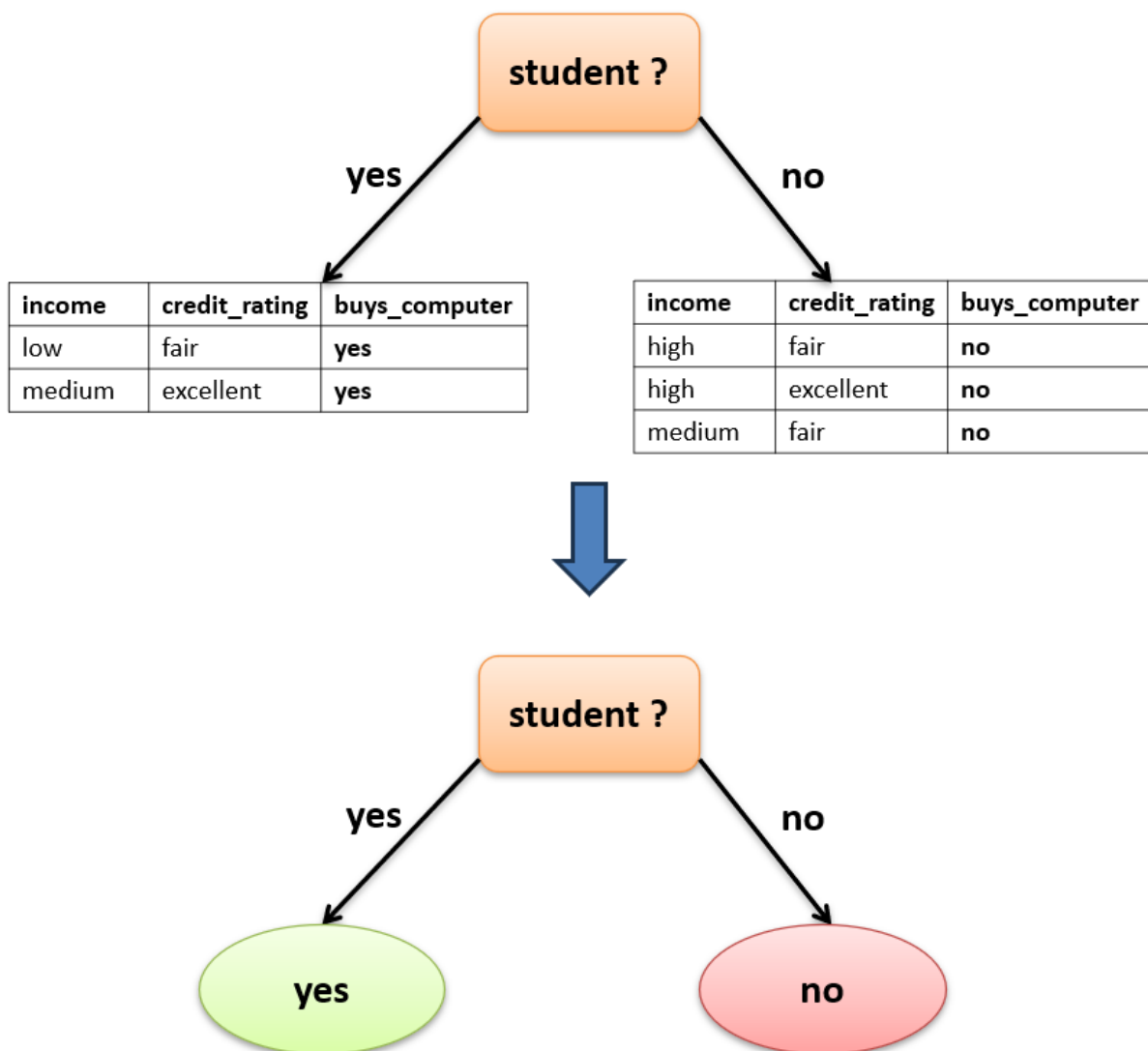


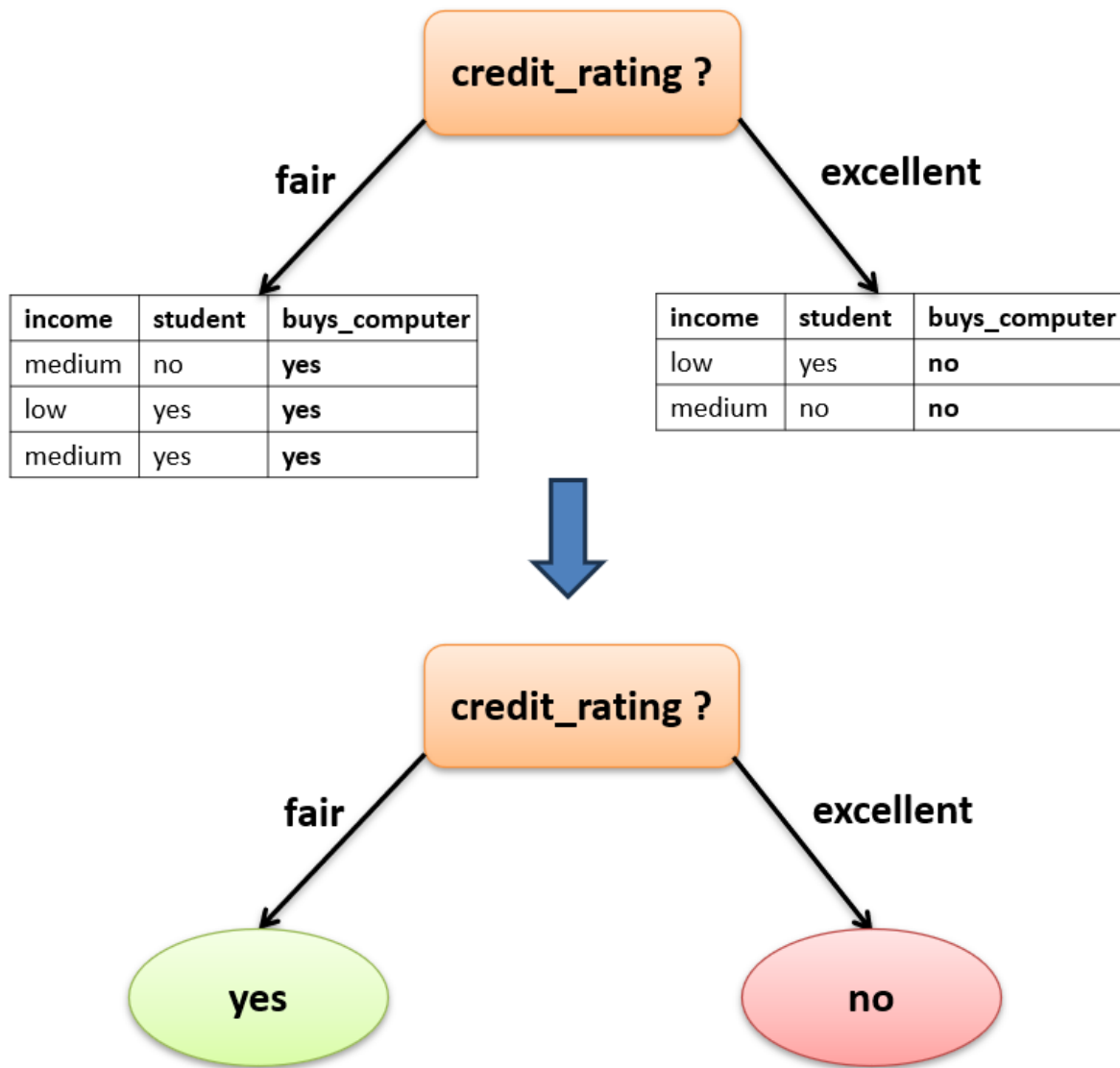
Fig: Decision tree for **student** attribute.

Notice that the tuples falling into the partition for **student** = **yes** all belong to the same class. Because they all belong to class “**yes**,” a leaf should therefore be created at the end of this branch and labelled “**yes**.” And the partition for **student** = **no** all belong to the same class therefore be created at the end of this branch and labelled “**no**.”

Let’s continue the same process for the **age** is “**senior**”.

If you continue the above process then we will get the attribute **credit_rating** has the *highest information gain* and therefore becomes the *splitting attribute* at the next level node of the decision

tree. Branches are grown for each outcome of **student**. The tuples are shown partitioned accordingly.



*Fig: Decision tree for **credit_rating** attribute.*

Now, we will reach to leaf nodes of decision tree. i.e., we have derived class attributes for all the attributes in the dataset.

The final *decision tree* for all the attributes in the above dataset using *ID3 algorithm*.

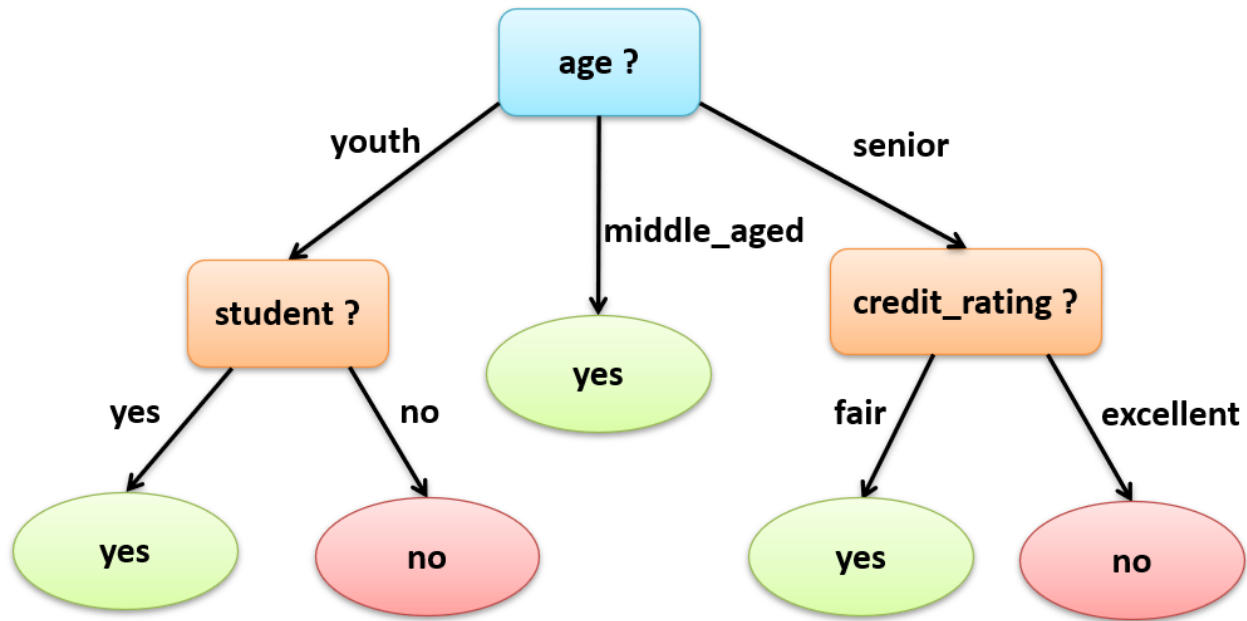


Fig: Decision tree

Exercise:

Let us illustrate the *ID3 algorithm* with a simple example of classifying *golf play* based on weather conditions. (or) Construct a *decision tree* to classify “*golf play*.”

Consider the following dataset:

Outlook	Temperature	Humidity	Wind	Golf_Play
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

Bayesian classification:

Bayesian classification is a *statistical technique* used to classify the data based on ***probabilistic reasoning***. It is a type of probabilistic classification that uses ***Bayes' theorem*** to predict the probability of a data point belonging to a ***certain class***.

Bayesian classification is a statistical method in data mining to data classification that uses **Bayes'theorem** to make ***predictions*** about a class of a data point based on observed data.

The basic idea behind Bayesian classification is ***to assign a class label*** to a new data record based on the probability that it belongs to a particular class.

The Bayesian classification is a powerful technique for probabilistic inference and decision-making and is widely used in various applications such as *medical diagnosis, spam classification, fraud detection*, etc.

Naïve Bayesian classifier:

Bayesian classification is based on Bayes' theorem, a simple Bayesian classifier known as the ***naïve Bayesian classifier***.

Bayes' theorem:

Bayes' theorem is named after ***Thomas Bayes***, an 18th-century British mathematician who first formulated it.

In data mining, *Bayes' theorem* is used to *compute* the probability of a hypothesis (such as a class label) for given some observed event (such as a set of attributes or features).

Bayes' theorem is a technique for predicting the class label of a new tuple / instance based on the probabilities of different class labels and the observed attributes/ features of the instance.

Bayes' Theorem consists of two types of probabilities: the *posterior probability* ($P(H/X)$) and the *prior probability* ($P(H)$), where X represents the data tuple and H represents a hypothesis.

The ***formula for Bayes' Theorem*** is as follows:

$$P(H|X) = \frac{P(X|H) * P(H)}{P(X)}$$

Let $\mathbf{X}$ is a *data tuple*. In Bayesian terms, $\mathbf{X}$ is considered “*evidence*.” As usual, it is described by measurements made on a set of n attributes. Let $\mathbf{H}$ be *some hypothesis*, such as that the data tuple $\mathbf{X}$ belongs to a specified *class C*.

For classification problems, we want to determine $P(\mathbf{H}/\mathbf{X})$, the probability that the hypothesis $\mathbf{H}$ holds given the observed data tuple $\mathbf{X}$. In other words, we are looking for the probability that tuple $\mathbf{X}$ belongs to *class C*, given that we know the attribute description of $\mathbf{X}$.

Posterior probability:

$P(\mathbf{H}/\mathbf{X})$ is the *posterior probability* or a posteriori probability of hypothesis $\mathbf{H}$ conditioned on data tuple $\mathbf{X}$.

For example, suppose we have data tuples from the customers database described by the attributes *age* and *income*, respectively, and that $\mathbf{X}$ is a *30-year-old* customer with an income of \$40,000. Suppose that $\mathbf{H}$ is the hypothesis that our customer will buy a computer. Then the $P(\mathbf{H}/\mathbf{X})$ represents the probability of customer $\mathbf{X}$ purchasing a computer, considering the customer's *age* and *income* information.

Similarly, $P(\mathbf{X}/\mathbf{H})$ is the *posterior probability* of $\mathbf{X}$ conditioned on hypothesis $\mathbf{H}$. That is, it is the probability that a customer, $\mathbf{X}$, is 30 years old and earns \$40,000, given that we know the customer will buy a computer.

Prior probability:

$P(\mathbf{H})$ is the **prior probability**, or *a priori probability*, of $\mathbf{H}$.

For example, this is the probability that any given customer will *buy a computer*, regardless of age, income, or any other information, for that matter. i.e., The *prior probability* $P(\mathbf{H})$, is independent of data tuple $\mathbf{X}$.

Similarly, $P(\mathbf{X})$ is the *prior probability* of $\mathbf{X}$. Using our example, it is the probability that a person from our data set of customers is 30 years old and earns \$40,000.

Naive Bayes Classifier Algorithm:

It is a classification technique based on **Bayes' Theorem** with an independence assumption among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.

For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as '**Naive**'.

Working of Naïve Bayes' Classifier:

Example:

Suppose we have a dataset of *weather conditions* and corresponding target variable "**Play**". So, using this dataset we need to decide that whether we should **play** or **not** on a particular day according to the **weather conditions**. So, to solve this problem, we need to follow the below steps:

- Convert the given dataset into **frequency tables**.
- Generate **Likelihood table** by finding the probabilities of given features.
- Now, use Bayes theorem to calculate the **posterior probability**.

Problem: If the Weather is **Sunny**, then the Player should **play** or **not**?

Solution: To solve this, first consider the below dataset:

	Weather	Humidity	Play
1	Rainy	High	Yes
2	Sunny	High	Yes
3	Overcast	High	Yes
4	Overcast	High	Yes
5	Sunny	Normal	No
6	Rainy	Normal	Yes
7	Sunny	Normal	Yes
8	Overcast	High	Yes
9	Rainy	Normal	No
10	Sunny	Normal	No
11	Sunny	Normal	Yes
12	Rainy	High	No
13	Overcast	Normal	Yes
14	Overcast	High	Yes

First, Convert the above table into *frequency table* as follows,

Weather	Yes	No
Overcast	5	0
Rainy	2	2
Sunny	3	2
Total	10	4

Next, Generate *Likelihood table* by finding the probabilities of given features as follows,

Weather	Yes	No	P(Yes)	P(No)	P
Overcast	5	0	5 / 10	0 / 4	5/14= 0.35
Rainy	2	2	2 / 10	2 / 4	4/14= 0.29
Sunny	3	2	3 / 10	2 / 4	5/14= 0.35
All	10/14= 0.71	4/14= 0.29			

Applying Bayes'theorem:

If the data tuple(**X**) is *Sunny*,and the**hypothesis(H)** is *Yes*then the probability of**Play is**:

$$P(\text{Yes}|\text{Sunny}) = \frac{P(\text{Sunny}|\text{Yes}) * P(\text{Yes})}{P(\text{Sunny})}$$

Where,

$$P(\text{Sunny}|\text{Yes}) = 3/10 = \mathbf{0.3}$$

$$P(\text{Yes}) = 10/14 = \mathbf{0.71}$$

$$P(\text{Sunny}) = 5/14 = \mathbf{0.35}$$

Let us substitute the values in the above formula,

$$\begin{aligned}
 P(\text{Yes}|\text{Sunny}) &= \frac{0.3 * 0.71}{0.35} \\
 &= \mathbf{0.60}
 \end{aligned}$$

If the Weather is *Sunny*,then the probability of**Play (Yes) is 0.60**

If the data tuple(**X**) is **Sunny**, and the hypothesis(**H**) is **No** then the probability of **Play** is:

$$P(\text{No}|\text{Sunny}) = \frac{P(\text{Sunny}|\text{No}) * P(\text{No})}{P(\text{Sunny})}$$

Where,

$$P(\text{Sunny}|\text{No}) = 2/4 = \mathbf{0.5}$$

$$P(\text{No}) = 4/14 = \mathbf{0.29}$$

$$P(\text{Sunny}) = 5/14 = \mathbf{0.35}$$

Let us substitute the values in the above formula,

$$\begin{aligned} P(\text{No}|\text{Sunny}) &= \frac{0.5 * 0.29}{0.35} \\ &= \mathbf{0.41} \end{aligned}$$

If the Weather is **Sunny**, then the probability of **Play (No)** is **0.41**.

So, as we can see from the above calculation that **P(Yes|Sunny) > P(No|Sunny)**

Hence on a **Sunny** day, Player can **play the game**.

Exercise:

Problem: If the Weather is **Sunny** and Humidity is **High**, then the Player should **play** or **not**?

Solution:

If the data tuple(**X**) is (**Sunny, High**), and the hypothesis(**H**) is **Yes** then the probability of **Play** is:

$$P(\text{Yes}|\text{(Sunny, High)}) = \frac{P(\text{Sunny}|\text{Yes}) * P(\text{High}|\text{Yes}) * P(\text{Yes})}{P(\text{Sunny}) * P(\text{High})}$$

If the data tuple(**X**) is (**Sunny, High**), and the hypothesis(**H**) is **No** then the probability of **Play** is:

$$P(\text{No}|\text{(Sunny, High)}) = \frac{P(\text{Sunny}|\text{No}) * P(\text{High}|\text{No}) * P(\text{No})}{P(\text{Sunny}) * P(\text{High})}$$

Rule-Based Classification:

Rule-based classification in data mining is a technique in which the records are *classified / categorized* by using a set of “**IF...THEN...**” rules.

Rules are a good way of representing information or bits of knowledge. A *rule-based classifier* uses a set of **IF-THEN** rules for *classification*.

An *IF-THEN rule* is an expression of the form

IF *condition* **THEN** *conclusion*.

An example is rule **R1**,

R1: IF *age = youth* AND *student = yes* **THEN** *buys_computer = yes*.

The “**IF**” part (or left side) of a rule is known as the rule *antecedent* or *precondition*. The “**THEN**” part (or right side) is the rule *consequent*.

In the rule *antecedent*, the *condition* consists of *one or more* attribute tests (e.g., *age = youth* and *student = yes*) that are **logically ANDed**. The rule’s *consequent* contains a **class prediction** (in this case, we are predicting whether a customer will buy a computer). **R1** can also be written as

R1: $(age = youth) \wedge (student = yes) \Rightarrow (buys_computer = yes)$

If the *condition* (i.e., all the attribute tests) in a *rule antecedent* holds *true* for a given tuple, we say that the rule antecedent is *satisfied* (or simply, that the rule is satisfied) and that the rule **covers** the tuple.

A rule **R** can be assessed by its *coverage* and *accuracy*.

Coverage: A rule’s coverage is the percentage of tuples that are **covered** by the rule.

Accuracy: A rule’s coverage is the percentage of tuples that are **correctly classified** by the rule.

Given a tuple, **X**, from a class labelled data set, **D**, let n_{covers} be the number of tuples covered by **R**; $n_{correct}$ be the number of tuples correctly classified by **R**; and $|D|$ be the number of tuples in **D**. We can define the **coverage** and **accuracy** of rule **R** as

$$\text{Coverage}(\mathbf{R}) = \frac{n_{covers}}{|D|}$$

$$\text{Accuracy}(\mathbf{R}) = \frac{n_{correct}}{n_{covers}}$$

Example: Rule **accuracy** and **coverage**. Let's go back to our data set in Table-1. These are class labelled tuples from the customer database. Our task is to predict whether a customer will buy a computer.

Consider rule **R1**, which covers **2** of the **14** tuples. It can correctly classify **both** tuples. Therefore, **coverage** R1 (D) = $2/14 = 14.28\%$ and **accuracy** R1 (D) = $2/2 = 100\%$.

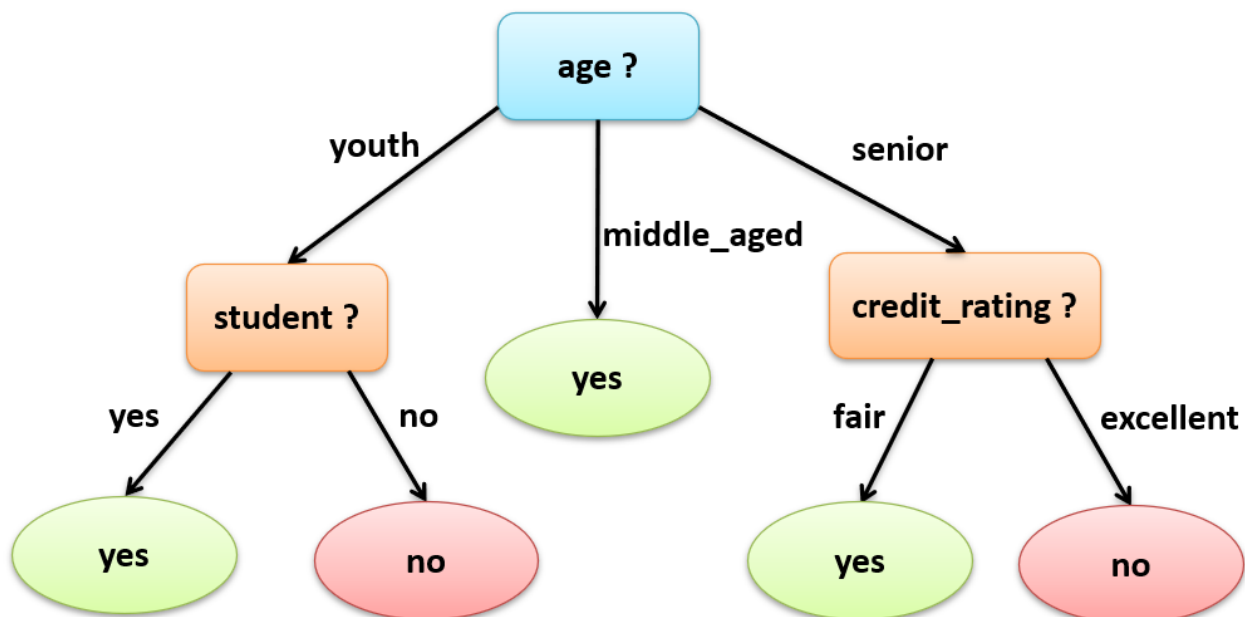
Rule Extraction from a Decision Tree:

Decision tree classifiers are a popular method of classification - it is easy to understand how decision trees work and they are known for their accuracy. Decision trees can become large and difficult to interpret.

In this, we look at how to build a **rule based classifier** by extracting **IF-THEN** rules from a **decision tree**. In comparison with a decision tree, the IF-THEN rules may be easier for humans to understand, particularly if the decision tree is very large.

To extract rules from a decision tree, one rule is created for each path from the **root** to a **leaf** node. Each splitting criterion along a given path is logically **ANDed** to form the rule antecedent ("IF" part). The leaf node holds the **class prediction**, forming the rule consequent ("THEN" part).

Example: Extracting **classification rules** from the following decision tree.



The decision tree of the above Figure can be converted to **classification IF-THEN rules** by tracing the path from the root node to each leaf node in the tree.

The rules extracted from the above Figure are as follows:

R1: IF *age* = *youth* AND *student* = *yes* THEN *buys_computer* = *yes*

R2: IF *age* = *youth* AND *student* = *no* THEN *buys_computer* = *no*

R3: IF *age* = *middle_aged* THEN *buys_computer* = *yes*

R4: IF *age* = *senior* AND *credit_rating* = *excellent* THEN *buys_computer* = *yes*

R5: IF *age* = *senior* AND *credit_rating* = *fair* THEN *buys_computer* = *no*

Lazy Learners (or Learning from Your Neighbors)

The classification methods discussed so far decision tree induction, Bayesian classification and rule-based classifications are all examples of **eager learners**.

Eager learners, when given a set of training tuples, will construct a classification model before receiving new (e.g., test) tuples to classify. We can think of the model as being ready and eager to classify previously unseen tuples.

Lazy learners simply store training tuples (or perform minimal processing) and wait until a test tuple is encountered. Only when it sees the **test tuple** does it perform generalization to classify the tuple based on its similarity to the stored training tuples.

Unlike eager learning methods, lazy learners do **less work** when a *training tuple* is presented and **more work** when making a *classification or numeric prediction* for the test tuple.

When making a *classification or numeric prediction*, lazy learners can be **computationally expensive**. They require efficient storage techniques and are well suited to implementation on parallel hardware.

The most popular **example** of Lazy learners is ***k-Nearest-Neighbor Classifiers***.

k-Nearest-Neighbor (KNN) Algorithm / Classifier:

The K-Nearest Neighbors (K-NN) algorithm is a popular algorithm used mostly for solving classification problems.

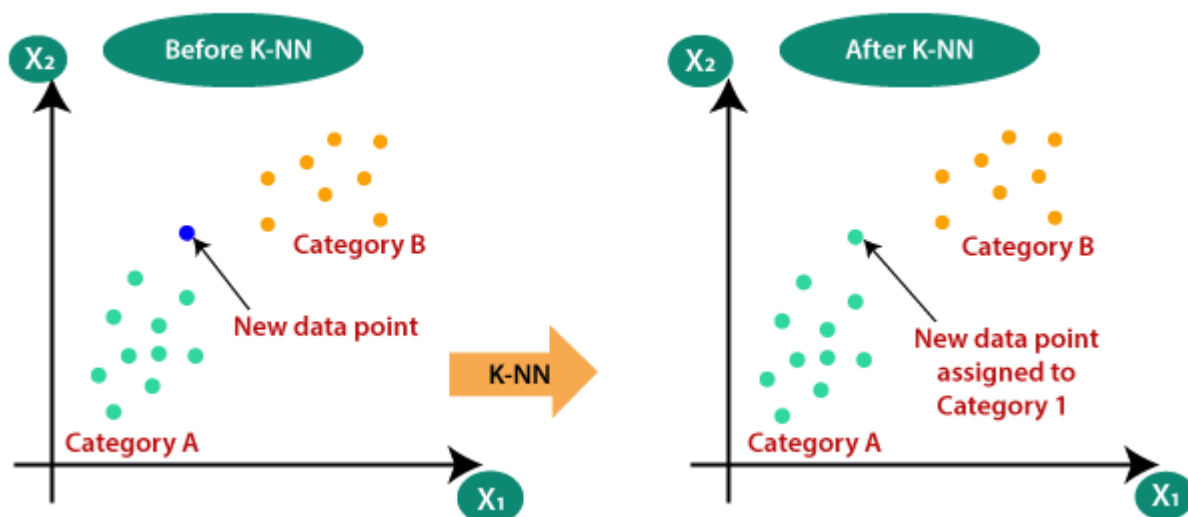
The K-NN algorithm stores all the available data (training data) and classifies a new data point (test data) based on the similarity.

In other words, The K-NN algorithm compares a new data point to the data / values in a given data set (with different classes or categories). Based on its *closeness or similarities* in a given range (K) of neighbors, the algorithm assigns the new data to a *class or category* in the data set (training data).

It is also called a **lazy learner algorithm** because it does not learn from the training set immediately instead it stores the dataset and at the time of classification, it performs an action on the dataset.

Why do we need a K-NN Algorithm?

Suppose we have two categories, namely *Category A* and *Category B*. Now, let's consider a new data point, **p1**. The question arises, which category does this data point belong to? To address this problem, we can utilize the K-NN algorithm. By employing the K-NN algorithm, we can easily identify the category or class of a particular dataset.



How k-Nearest-Neighbor (K-NN) Algorithm works?

The algorithm consists of the subsequent steps.

Step -1: Choose the value K for the neighbors.

Step -2: Compute the distance between the new data point and all other existing data points. And arrange them in ascending order.

Step -3: Find the K nearest neighbors to the new data point based on the computed distances.

Step -4: Count the number of data points in each category among these k neighbors.

Step -5: Assign the new data point to that class / category for which the number of the neighbors is maximum.

How to Choose the Value of K in the K-NN Algorithm?

There is *no specific way* of choosing the value **K**, but here are some common conventions to keep in mind:

- Choosing a very low value will most likely lead to inaccurate predictions.
- The commonly used value of **K** is **5**.
- Always use an odd number as the value of K.
- If the input data has more outliers or noise, a higher value of k would be better.

How to Compute the Distance? (or) Distance Measures Used in KNN Algorithm:

The K-NN algorithm enables us to classify or categorize the new data point by *calculating the distance* from other pre-existing data points. To calculate the distance we use any of the following are *Distance Measures*.

Euclidean distance:

Euclidean distance can be simply explained as the *ordinary distance* between two points. Mathematically it computes the *root of squared differences* between the coordinates between two objects.

The *Euclidean distance* between two points or tuples, say, **X** = (x1, x2, . . . , xn) and **Y** = (y1, y2, . . . , yn) is

$$dist(X,Y) = dist(Y,X) = \sqrt{\sum_{i=1}^n (X_i - Y_i)^2}$$

Manhattan distance:

The Manhattan distance, also called the *Taxicab distance* or the *City Block distance*, calculates the distance between two real-valued vectors.

Manhattan Distance metric is generally used when we are interested in the total distance travelled by the object instead of the displacement. This metric is calculated by ***summing the absolute difference*** between the coordinates of the points in n-dimensions.

The ***Manhattan distance*** between two points or tuples, say, $\mathbf{X} = (x_1, x_2, \dots, x_n)$ and $\mathbf{Y} = (y_1, y_2, \dots, y_n)$ is

$$\text{dist}(\mathbf{X}, \mathbf{Y}) = \text{dist}(\mathbf{Y}, \mathbf{X}) = \sum_{i=1}^n |X_i - Y_i|$$

Example:

Let $\mathbf{X} = (1, 2)$ and $\mathbf{Y} = (3, 5)$ represent two data points.

The ***Euclidean distance*** between the two data points is

$$\begin{aligned}\text{dist}(\mathbf{X}, \mathbf{Y}) &= \sqrt{(1 - 3)^2 + (2 - 5)^2} \\ &= \sqrt{(-2)^2 + (-3)^2} \\ &= \sqrt{4 + 9} \\ &= \sqrt{13} = \mathbf{3.61}\end{aligned}$$

The ***Manhattan distance*** between the two data points is

$$\begin{aligned}\text{dist}(\mathbf{X}, \mathbf{Y}) &= |1 - 3| + |2 - 5| \\ &= 2 + 3 = \mathbf{5}\end{aligned}$$

Note:

Euclidean distance is the most popular distance measure to calculate the distance between data points or tuples.

Example:

Consider the following training dataset,

Height (in CM)	Weight(in KG)	Class
189	87	Normal
149	61	Overweight
189	104	Overweight
195	81	Normal
155	51	Normal
191	79	Normal
190	95	Overweight
157	56	Normal
185	76	Normal
161	72	Overweight
164	75	Overweight
149	66	Overweight

And apply **K-NN algorithm** to find the *class label* for the following test tuple.

Height (in CM)	Weight(in KG)	Class
148	60	?

Solution:

In the given dataset, we have *two columns* – *Height* (in CM) and *Weight* (in KG). Each row in the dataset has a class of either *Normal* or *Overweight*.

Step-1:

To start working with K-NN algorithm, first we need to choose the value for **K**.

Let's assume the value of **K** is **5**.

Step-2:

In this, we need to calculate the distance between *new (test) tuple* and all other tuples in the *existing (training)* dataset.

To calculate the distance we choose *Euclidean distance measure*.

Here's the *new (test) data entry*:

Height (in CM)	Weight(in KG)	Class
148	60	?

Now, we need to calculate the distance from this new tuple to all other data tuples.

Here's the formula:

$$\sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}$$

Where:

- X_2 = New entry's Height (148).
- X_1 = Existing entry's Height.
- Y_2 = New entry's Weight (60).
- Y_1 = Existing entry's Weight.

Let's start the calculation

Distance #1

For the first row, **d1**:

Height (in CM)	Weight(in KG)	Class
189	87	Normal

$$\begin{aligned} \mathbf{d1} &= \sqrt{(148 - 189)^2 + (60 - 87)^2} \\ &= \sqrt{1681 + 729} \\ &= \sqrt{2410} \\ &= 49.1 \end{aligned}$$

Now, we know the distance from the new data entry/tuple. Let's update the table/dataset.

Height (in CM)	Weight(in KG)	Class	Distance
189	87	Normal	49.09

Distance #2

For the first row, **d2**:

Height (in CM)	Weight(in KG)	Class
149	61	Overweight

$$\begin{aligned} \mathbf{d2} &= \sqrt{(148 - 149)^2 + (60 - 61)^2} \\ &= \sqrt{2} \\ &= 1.41 \end{aligned}$$

Now, we know the distance from the new data entry/tuple. Let's update the table/dataset.

Height (in CM)	Weight(in KG)	Class	Distance
149	61	Overweight	1.41

Distance #3

For the first row, **d3**:

Height (in CM)	Weight(in KG)	Class
189	104	Overweight

$$\begin{aligned}d3 &= \sqrt{(148 - 189)^2 + (60 - 104)^2} \\&= \sqrt{1681 + 1936} \\&= 60.14\end{aligned}$$

Now, we know the distance from the new data entry/tuple. Let's update the table/dataset.

Height (in CM)	Weight(in KG)	Class	Distance
149	61	Overweight	60.14

Like above we have to calculate the distance from all the other data entries/tuples.

After calculating the distances from all other existing data tuples, the updated table or dataset is

Height(in CM)	Weight(in KG)	Class	Distance
189	87	Normal	49.09
149	61	Overweight	1.41
189	104	Overweight	60.14
195	81	Normal	51.48
155	51	Normal	11.40
191	79	Normal	47.01
190	95	Overweight	54.67
157	56	Normal	9.85
185	76	Normal	40.31
161	72	Overweight	17.69
164	75	Overweight	21.93
149	66	Overweight	6.08

Let's *rearrange* the distances *in ascending order*:

Height(in CM)	Weight(in KG)	Class	Distance
149	61	Overweight	1.41
149	66	Overweight	6.08
157	56	Normal	9.85
155	51	Normal	11.40
161	72	Overweight	17.69
164	75	Overweight	21.93
185	76	Normal	40.31
191	79	Normal	47.01
189	87	Normal	49.09
195	81	Normal	51.48
190	95	Overweight	54.67
189	104	Overweight	60.14

Step -3:

Since we chose 5 as the value of **K**, we'll only consider the *first five rows*.

Height(in CM)	Weight(in KG)	Class	Distance
149	61	Overweight	1.41
149	66	Overweight	6.08
157	56	Normal	9.85
155	51	Normal	11.40
161	72	Overweight	17.69

Step -4:

In the above 5 nearest neighbors, **3** neighbors have ***Overweight*** as a class and **2** neighbours have ***Normal*** as a class.

Step -5:

As you can see above, the class ***Overweight*** has maximum number of neighbors, Therefore, we'll classify the new entry/tuple as ***Overweight***.

The **updated tuple** is

Height (in CM)	Weight(in KG)	Class
148	60	Overweight

Advantages of K-NN Algorithm

- It is simple to implement.
- No training is required before classification.

Disadvantages of K-NN Algorithm

- Can be cost-intensive when working with a large data set.
- A lot of memory is required for processing large data sets.
- Choosing the right value of K can be tricky.

Exercise:

Consider the following training dataset,

Brightness	Saturation	Class
40	20	Red
50	50	Blue
60	90	Blue
10	25	Red
70	70	Blue
60	10	Red
25	80	Blue

And apply **K-NN algorithm** to find the *class label* for the following test tuple.

Brightness	Saturation	Class
20	36	?