INDERPRASTHA ENGINEERING COLLEGE GHAZIABAD

Department of Information Technology

Data Warehousing and Data Mining Lab (RCS-654)

Name	**: Natasha Sharma**
Roll Number	**: 1703013043**
Course	**: B.TECH**
Year	**: 3rd**
Semester	**: 6th**
Section	**: A**

INDEX

<u>Program-1</u>

Aim: Explore the tool WEKA and create dataset used in WEKA.

Theory:

Weka is a collection of machine learning algorithms for data mining tasks. It contains tools for data preparation, classification, regression, clustering, association rules mining, and visualization.Found only on the islands of New Zealand, the Weka is a flightless bird with an inquisitive nature. **Weka** is a collection of machine learning algorithms for data mining tasks. The algorithms can either be applied directly to a dataset or called from your own Java code. **Weka** contains tools for data pre-processing, classification, regression, clustering, association rules, and visualization.**Weka** is open source software issued under the GNU General Public License.

Various tasks performed by WEKA

1. Preprocess. Choose and modify the data being acted on.
2. Classify. Train and test learning schemes that classify or perform regression.
3. Cluster. Learn clusters for the data.
4. Associate. Learn association rules for the data.
5. Select attributes. Select the most relevant attributes in the data.
6. Visualize. View an interactive 2D plot of the data.

Dataset format in WEKA

An ARFF (Attribute-Relation File Format) file is an ASCII text file that describes a list of instances sharing a set of attributes. ARFF files were developed by the Machine Learning Project at the Department of Computer Science of The University of Waikato for use with the Weka machine learning software.

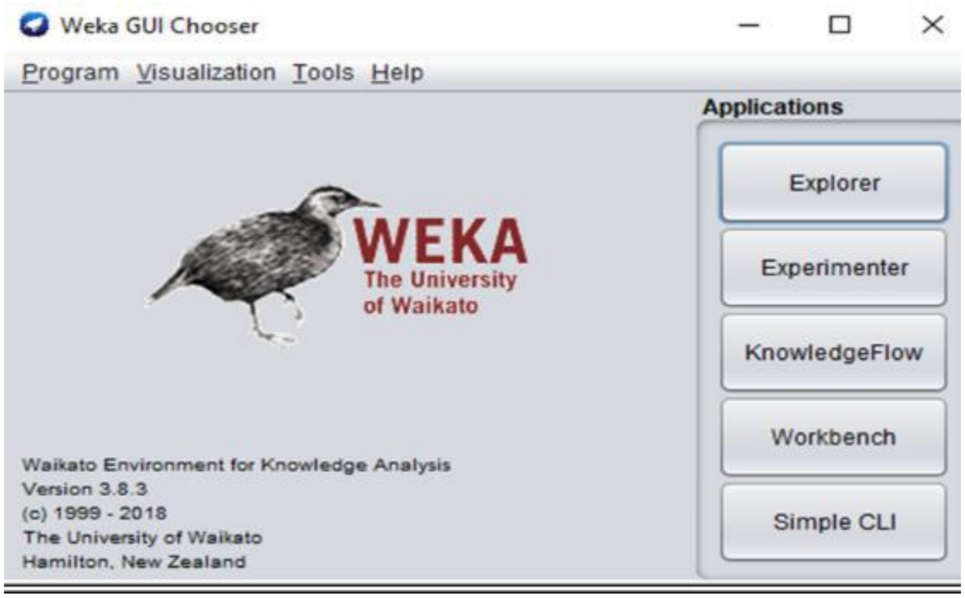
ARFF files have two distinct sections. The first section is the Header information, which is followed the Data information. The Header of the ARFF file contains the name of the relation, a list of the attributes (the columns in the data), and their types.

Example of the dataset in ARFF format

```
@relation weather
@attribute outlook {sunny, overcast, rainy}
@attribute temperature numeric
@attribute humidity numeric
@attribute windy {TRUE, FALSE}
@attribute play {yes, no}
@data
sunny,85,85,FALSE,no
sunny,80,90,TRUE,no
overcast,83,86,FALSE,yes
rainy,70,96,FALSE,yes
```

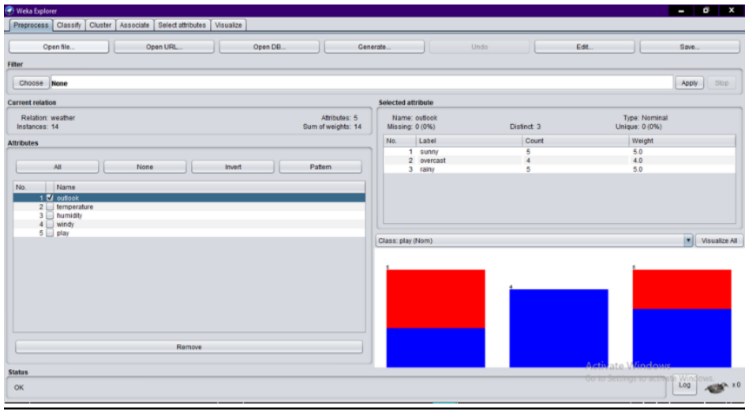
```
rainy,68,80,FALSE,yes
rainy,65,70,TRUE,no
overcast,64,65,TRUE,yes
sunny,72,95,FALSE,no
sunny,69,70,FALSE,yes
rainy,75,80,FALSE,yes
sunny,75,70,TRUE,yes
overcast,72,90,TRUE,yes
overcast,81,75,FALSE,yes
rainy,71,91,TRUE,no
```

WEKA HOMESCREEN



Waikato Environment for Knowledge Analysis
Version 3.8.3
(c) 1999 - 2018
The University of Waikato
Hamilton, New Zealand

WEKA EXPLORER



Filter: None

Current relation
Relation: weather
Instances: 14
Attributes: 5
Sum of weights: 14

Attributes

No.		Name
1	✓	outlook
2		temperature
3		humidity
4		windy
5		play

Selected attribute
Name: outlook
Missing: 0 (0%)
Distinct: 3
Type: Nominal
Unique: 0 (0%)

No.	Label	Count	Weight
1	sunny	5	5.0
2	overcast	4	4.0
3	rainy	5	5.0

Class: play (Nom)

Status
OK

Program-2

Aim: Explore various filters used in the tool Weka.

Theory:

Pre-processing tools in WEKA are called “filters”. WEKA contains filters for discretization, normalization, resampling, attribute selection, transformation and combination of attributes. The weka.filters package is concerned with classes that transform datasets. This package offers useful support for data pre-processing, which is an important step in machine learning. All filters offer the options -i for specifying the input dataset, and -o for specifying the output dataset. If any of these parameters is not given, standard input and/or standard output will be read from/written to. Other parameters are specific to each filter and can be found out via -h, as with any other class.

The weka.filters package is organized into supervised and unsupervised filtering, both of which are again subdivided into instance and attribute filtering.

- weka.filters.supervised:
  It contains supervised filters, i.e., filters that take class distributions into account. Classes below weka.filters.supervised in the class hierarchy are for supervised filtering, i.e., taking advantage of the class information. A class must be assigned via -c, for WEKA default behaviour use -c last.
- weka.filters.unsupervised:
  It contains unsupervised filters, i.e., they work without taking any class distributions into account. Classes below weka.filters.unsupervised in the class hierarchy are for unsupervised filtering, e.g. the non-stratified version of Resample. A class attribute should not be assigned here.

Attribute - filters that work column-wise.
Instance - filters that work row-wise.

Procedure to apply filters:

To apply a filter, click Choose button under Filter. Click Filter button at the bottom of the drop-down window.

Various filters used in weka:

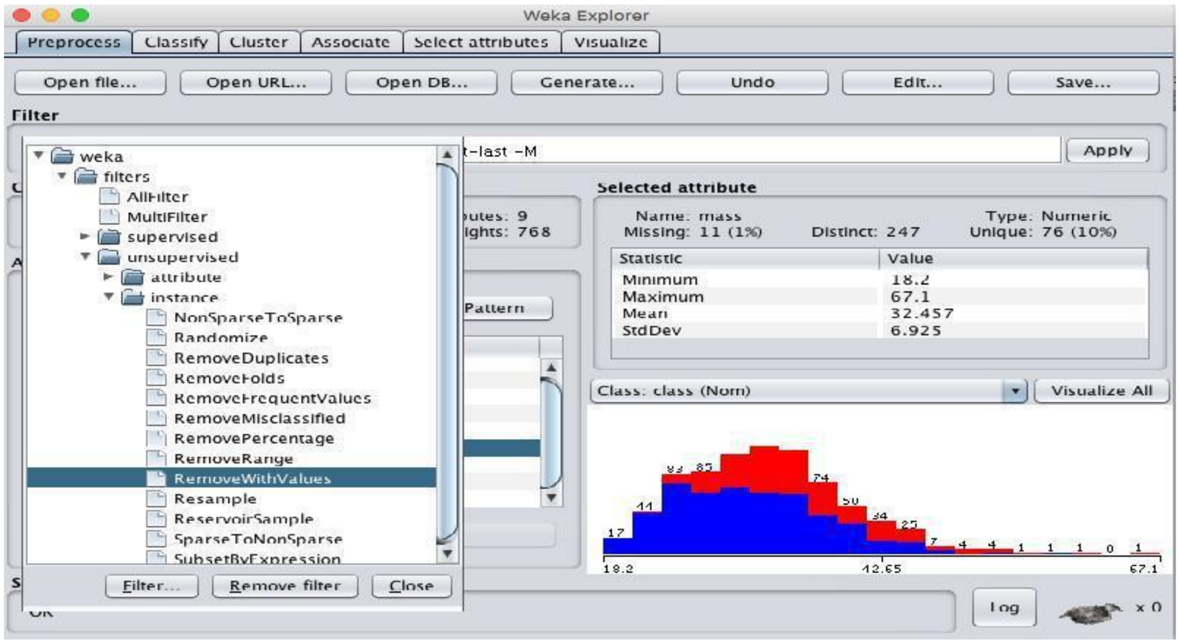
Supervised filters:-

- Attribute Selection-Filter for doing attribute selection.
- Class Order- A filter that sorts the order of classes so that the class values are no longer of in the order of that in the header file after filtered.
- Discretize- An instance filter that discretizes a range of numeric attributes in the dataset into nominal attributes.
- NominalToBinary- Converts all nominal attributes into binary numeric attributes.
- Resample- Produces a random subsample of a dataset.
- SpreadSubsample- Produces a random subsample of a dataset.
- StratifiedRemoveFolds- This filter takes a dataset and outputs folds suitable for cross validation.

<u>Unsupervised filters:-</u>

- Add- An instance filter that adds a new attribute to the dataset.
- AddCluster- A filter that adds a new nominal attribute representing the cluster assigned to each instance by the specified clustering algorithm.
- AddExpression- Applys a mathematical expression involving attributes and numeric constants to a dataset.
- AddNoise- Introduces noise data a random subsample of the dataset by changing a given attribute.
- Copy- An instance filter that copies a range of attributes in the dataset.
- MergeTwoValues- Merges two values of a nominal attribute.
- Normalize- Normalizes all numeric values in the given dataset.
- NumericToBinary- Converts all numeric attributes into binary attributes (apart from the class attribute): if the value of the numeric attribute is exactly zero, the value of the new attribute will be zero.
- RandomProjection- Reduces the dimensionality of the data by projecting it onto a lower dimensional subspace using a random matrix with columns of unit length.
- Remove- An instance filter that deletes a range of attributes from the dataset.
- ReplaceMissingValues- Replaces all missing values for nominal and numeric attributes in a dataset with the modes and means from the training data.
- Standardize- Standardizes all numeric attributes in the given dataset to have zero mean and unit variance.
- StringToNominal- Converts a string attribute (i.e. unspecified number of values) to nominal (i.e. set number of values).
- NonSparseToSparse- A filter that converts all incoming instances into sparse format.
- Randomize- This filter randomly shuffles the order of instances passed through it.
- RemoveRange- This filter takes a dataset and removes a subset
- RemoveWithValues- Filters instances according to the value of an attribute.
- SparseToNonSparse- A filter that converts all incoming sparse instances into non-sparse Format.

WEKA Filter



t-last -M

Selected attribute

Name: mass | Type: Numeric
Missing: 11 (1%) | Distinct: 247 | Unique: 76 (10%)

Statistic	Value
Minimum	18.2
Maximum	67.1
Mean	32.457
StdDev	6.925

<u>Program-3</u>

Aim: Demonstrate pre-processing of WEKA data-set.

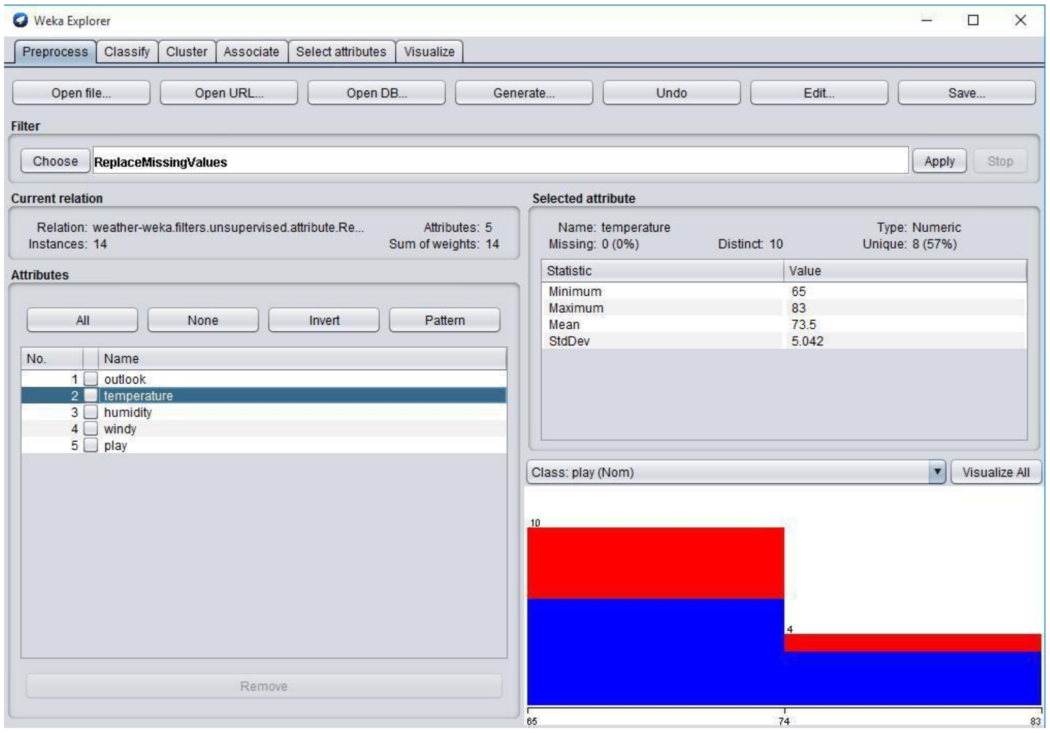
Theory:

The data that is collected from the field contains many unwanted things that leads to wrong analysis. For example, the data may contain null fields, it may contain columns that are irrelevant to the current analysis, and so on. Thus, the data must be preprocessed to meet the requirements of the type of analysis you are seeking. This is the done in the preprocessing module. To demonstrate preprocessing, we will use the **Weather** database.

Procedure for pre processing:

- **Understand the data.**

The weather database contains five fields - outlook, temperature, humidity, windy and play. There are 14 instances - the number of rows in the table.The table contains 5 attributes - the fields



Filter: ReplaceMissingValues

Current relation
Relation: weather-weka.filters.unsupervised.attribute.Re...   Attributes: 5
Instances: 14   Sum of weights: 14

Selected attribute
Name: temperature   Type: Numeric
Missing: 0 (0%)   Distinct: 10   Unique: 8 (57%)

Statistic	Value
Minimum	65
Maximum	83
Mean	73.5
StdDev	5.042

Attributes

No.	Name
1	outlook
2	temperature
3	humidity
4	windy
5	play

Class: play (Nom)

In the **Selected Attribute** sub window, we can observe

The name and the type of the attribute are displayed.

The type for the **temperature** attribute is **Numeric.**

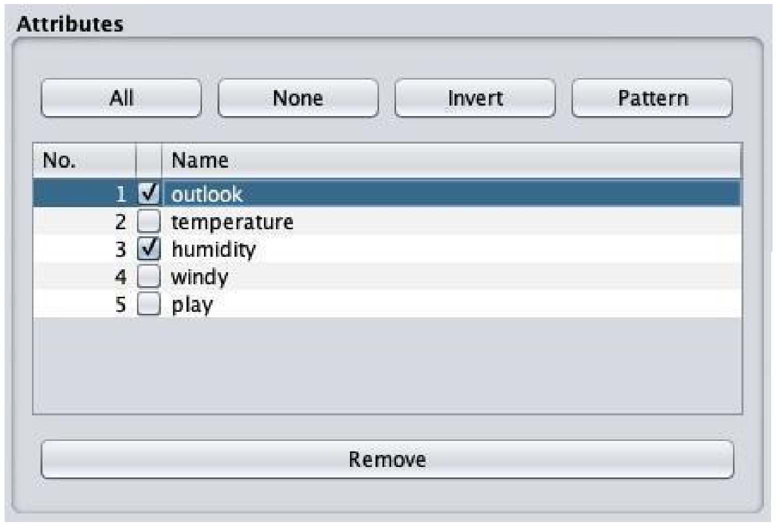
The number of **Missing** values is zero.

- ## Removing Attributes

Many a time, the data that you want to use for model building comes with many irrelevant fields. For example, the customer database may contain his mobile number which is relevant in analyzing his credit rating.

To remove Attribute/s select them and click on the **Remove** button at the bottom.

The selected attributes would be removed from the database. After you fully preprocess the data, you can save it for model building.

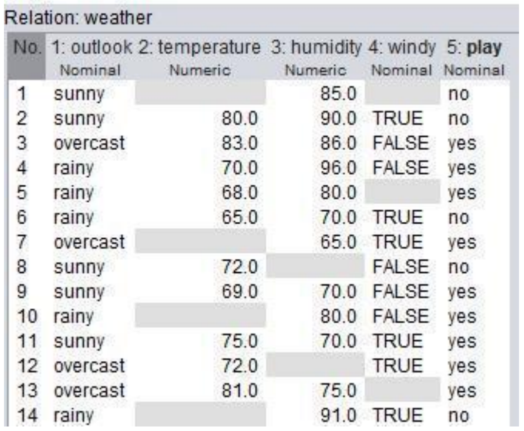


No.		Name
1	✓	outlook
2		temperature
3	✓	humidity
4		windy
5		play

- ## Replacing Missing values

Many times the data set has some null values or empty tuples in order to overcome that we need to substitute some values with the help of filters we can do so.

weka→filters→unsupervised→attribute→Replacemissingvalues



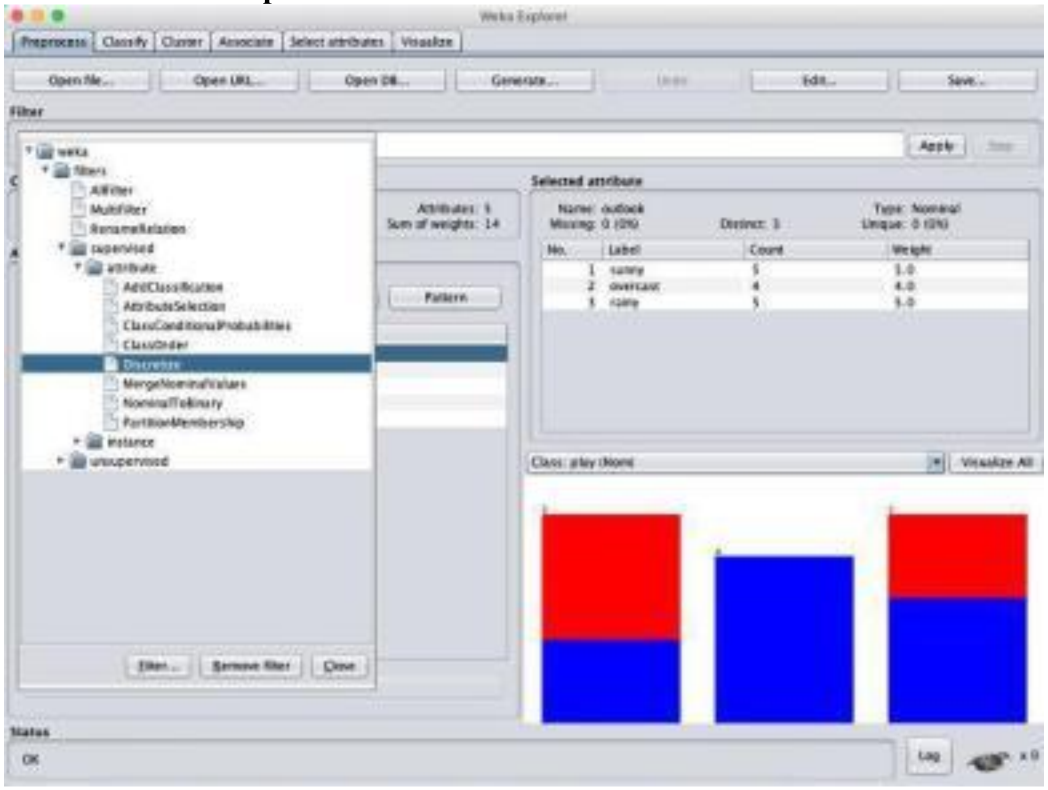
Relation: weather

No.	1: outlook Nominal	2: temperature Numeric	3: humidity Numeric	4: windy Nominal	5: play Nominal
1	sunny		85.0		no
2	sunny	80.0	90.0	TRUE	no
3	overcast	83.0	86.0	FALSE	yes
4	rainy	70.0	96.0	FALSE	yes
5	rainy	68.0	80.0		yes
6	rainy	65.0	70.0	TRUE	no
7	overcast		65.0	TRUE	yes
8	sunny	72.0		FALSE	no
9	sunny	69.0	70.0	FALSE	yes
10	rainy		80.0	FALSE	yes
11	sunny	75.0	70.0	TRUE	yes
12	overcast	72.0		TRUE	yes
13	overcast	81.0	75.0		yes
14	rainy		91.0	TRUE	no

- **Discretization**

Some association rules can only be performed on categorical data. This requires performing discretization on numeric or continuous attributes. The Data set contains two numeric attributes - temperature and humidity. We will convert these to nominal by applying a filter on our raw data.

weka→filters→supervised→attribute→Discretize



Attributes: 5
Sum of weights: 14

Name: outlook
Missing: 0 (0%)
Distinct: 3
Type: Nominal
Unique: 0 (0%)

No.	Label	Count	Weight
1	sunny	5	5.0
2	overcast	4	4.0
3	rainy	5	5.0

Program-4

Aim: Demonstration of Association rule process on dataset contactlenses.arff using apriori algorithm

Theory:

In data mining, association rule learning is a popular and well-accepted method for discovering interesting relations between variables in large databases. Association rules are employed today in many areas including web usage mining, intrusion detection and bioinformatics. The sample dataset used for this is contactlenses.arff.

```
@relation contact-lenses
@attribute age {young, pre-presbyopic, presbyopic}
@attribute spectacle-prescrip {myope, hypermetrope}
@attribute astigmatism{no, yes} @attribute tear-
prod-rate{reduced, normal}
@attribute contact-lenses{soft, hard, none}
```

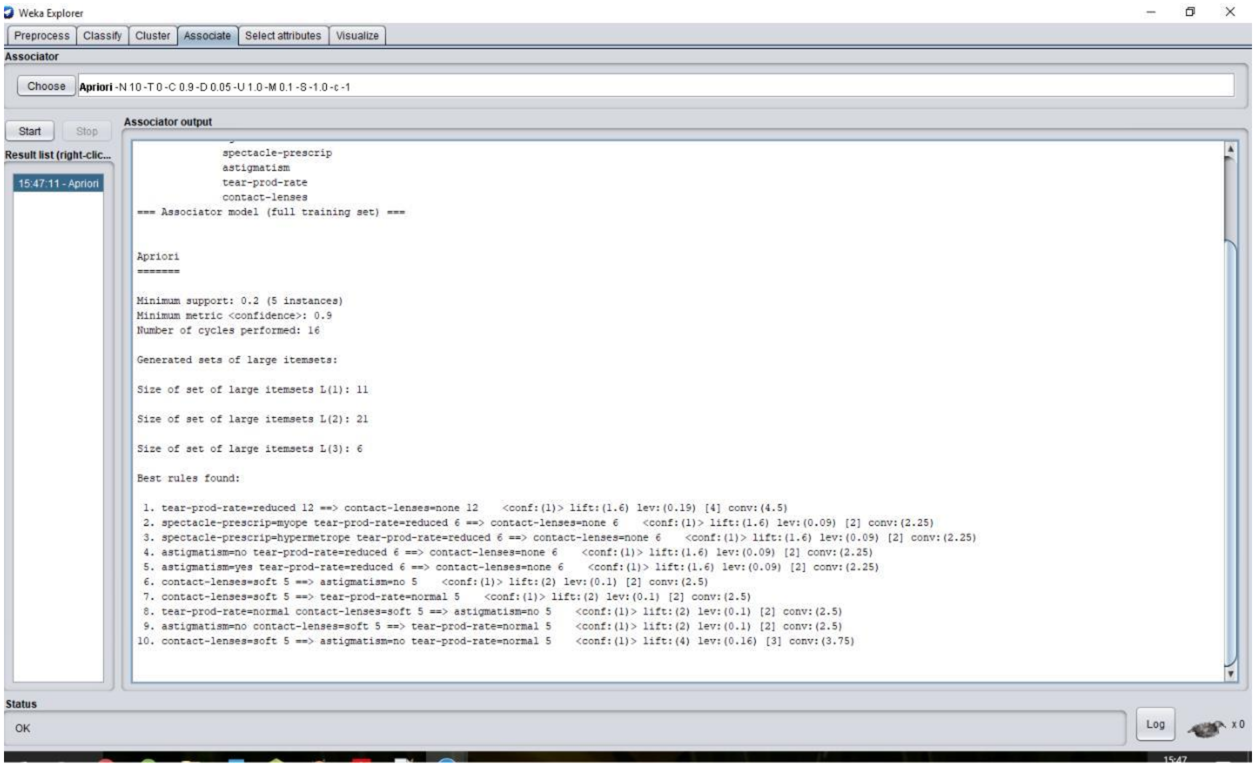
Dataset :

```
@data
young,myope,no,reduced,none
young,myope,no,normal,soft
young,myope,yes,reduced,none
young,myope,yes,normal,hard
young,hypermetrope,no,reduced,none
young,hypermetrope,no,normal,soft
young,hypermetrope,yes,reduced,none
young,hypermetrope,yes,normal,hard
pre-presbyopic,myope,no,reduced,none
pre-presbyopic,myope,no,normal,soft
pre-presbyopic,myope,yes,reduced,none
pre-presbyopic,myope,yes,normal,hard
pre-presbyopic,hypermetrope,no,reduced,none
pre-presbyopic,hypermetrope,no,normal,soft
pre-presbyopic,hypermetrope,yes,reduced,none
pre-presbyopic,hypermetrope,yes,normal,none
presbyopic,myope,no,reduced,none
presbyopic,myope,no,normal,none
presbyopic,myope,yes,reduced,none
presbyopic,myope,yes,normal,hard
presbyopic,hypermetrope,no,reduced,none
presbyopic,hypermetrope,no,normal,soft
presbyopic,hypermetrope,yes,reduced,none
presbyopic,hypermetrope,yes,normal,none
```

Steps Involved:

Step1: Open the data file in Weka Explorer. It is presumed that the required data fields have been discretized. In this example it is age attribute.
Step2: Clicking on the associate tab will bring up the interface for association rule algorithm.
Step3: We will use apriori algorithm. This is the default algorithm.
Step4: Inorder to change the parameters for the run (example support, confidence etc) we click on the text box immediately to the right of the choose button.

OUTPUT:



Apriori -N 10 -T 0 -C 0.9 -D 0.05 -U 1.0 -M 0.1 -S -1.0 -c -1

```
              spectacle-prescrip
              astigmatism
              tear-prod-rate
              contact-lenses
=== Associator model (full training set) ===


Apriori
=======

Minimum support: 0.2 (5 instances)
Minimum metric <confidence>: 0.9
Number of cycles performed: 16

Generated sets of large itemsets:

Size of set of large itemsets L(1): 11

Size of set of large itemsets L(2): 21

Size of set of large itemsets L(3): 6

Best rules found:

 1. tear-prod-rate=reduced 12 ==> contact-lenses=none 12    <conf:(1)> lift:(1.6) lev:(0.19) [4] conv:(4.5)
 2. spectacle-prescrip=myope tear-prod-rate=reduced 6 ==> contact-lenses=none 6    <conf:(1)> lift:(1.6) lev:(0.09) [2] conv:(2.25)
 3. spectacle-prescrip=hypermetrope tear-prod-rate=reduced 6 ==> contact-lenses=none 6    <conf:(1)> lift:(1.6) lev:(0.09) [2] conv:(2.25)
 4. astigmatism=no tear-prod-rate=reduced 6 ==> contact-lenses=none 6    <conf:(1)> lift:(1.6) lev:(0.09) [2] conv:(2.25)
 5. astigmatism=yes tear-prod-rate=reduced 6 ==> contact-lenses=none 6    <conf:(1)> lift:(1.6) lev:(0.09) [2] conv:(2.25)
 6. contact-lenses=soft 5 ==> astigmatism=no 5    <conf:(1)> lift:(2) lev:(0.1) [2] conv:(2.5)
 7. contact-lenses=soft 5 ==> tear-prod-rate=normal 5    <conf:(1)> lift:(2) lev:(0.1) [2] conv:(2.5)
 8. tear-prod-rate=normal contact-lenses=soft 5 ==> astigmatism=no 5    <conf:(1)> lift:(2) lev:(0.1) [2] conv:(2.5)
 9. astigmatism=no contact-lenses=soft 5 ==> tear-prod-rate=normal 5    <conf:(1)> lift:(2) lev:(0.1) [2] conv:(2.5)
10. contact-lenses=soft 5 ==> astigmatism=no tear-prod-rate=normal 5    <conf:(1)> lift:(4) lev:(0.16) [3] conv:(3.75)
```

Program-5

Aim: Demonstration of classification rule process on Weka data-set using j48 algorithm.

Theory:

Algorithm for building decision trees is implemented in Weka as a classifier called J48.Classifiers, like filters, are organized in a hierarchy. J48 has the full name weka.classifiers.trees J48. We have used student.arff dataset for applying the classifier rule which gives result if the student buys pc or not.

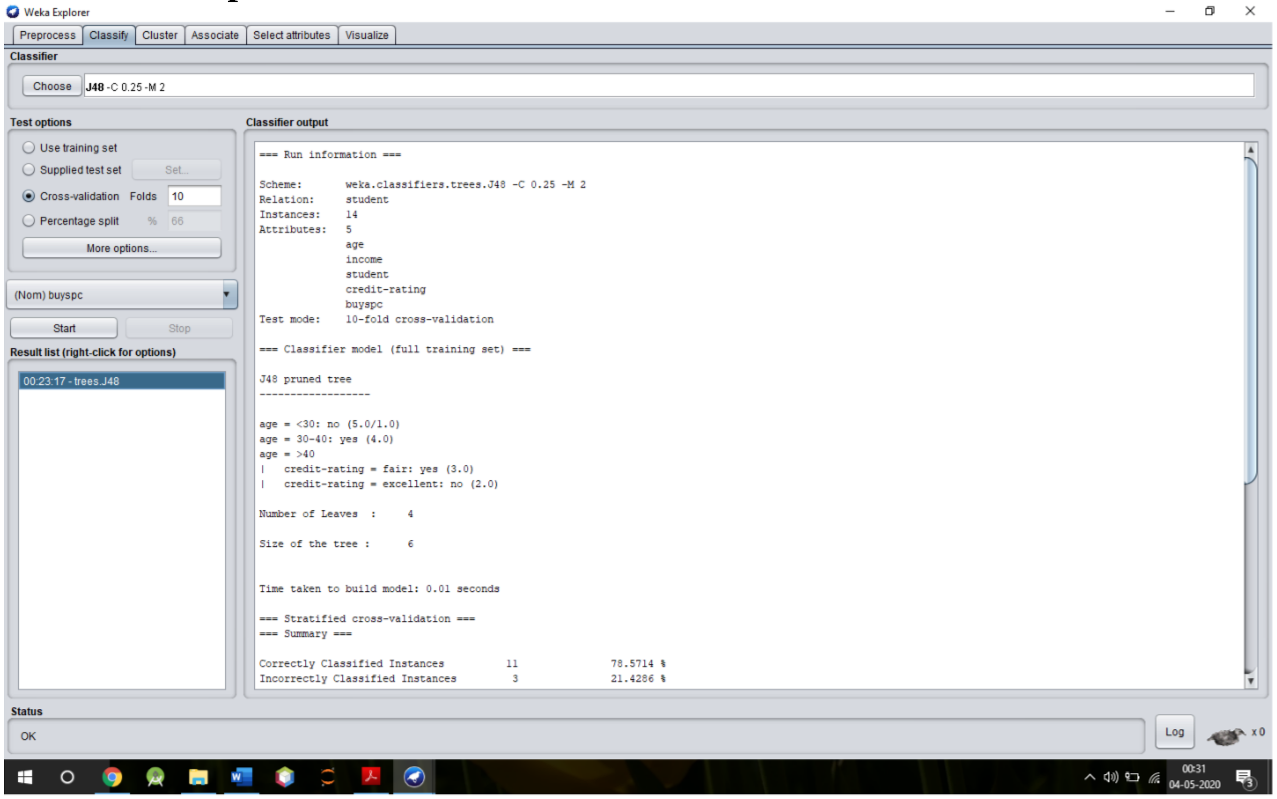
Dataset:

```
@relation student
@attribute age{<30,30-40,>40}
@attribute income {low, medium, high}
@attribute student {yes, no}
@attribute credit-rating {fair, excellent}
 @attribute buyspc {yes, no}
@data
%
<30, high, no, fair, no
<30, high, no, excellent, no
30-40, high, no, fair, yes
>40, medium, no, fair, yes
>40, low, yes, fair, yes
>40, low, yes, excellent, no
30-40, low, yes, excellent, yes
<30, medium, no, fair, no
<30, low, yes, fair, no
>40, medium, yes, fair, yes
<30, medium, yes, excellent, yes
30-40, medium, no, excellent, yes
30-40, high, yes, fair, yes
>40, medium, no, excellent, no
%
```

Steps Involved:

Step 1: Load the data (student.arff) into weka.
Step2: Select the "classify" tab and click "choose" button to select the "j48" classifier.
Step3: Specify the various parameters.
Step-4: We now click "start" to generate the model. the ASCII version of the tree as well as evaluation statistic will appear in the right panel when the model construction is complete.
Step-5: Weka also lets us a view a graphical version of the classification tree. This can be done by right clicking the last result set and selecting "visualize tree" from the pop-up menu.

OUTPUT:
Classifier Output



Classifier: J48 -C 0.25 -M 2

Test options: Cross-validation Folds 10

(Nom) buyspc

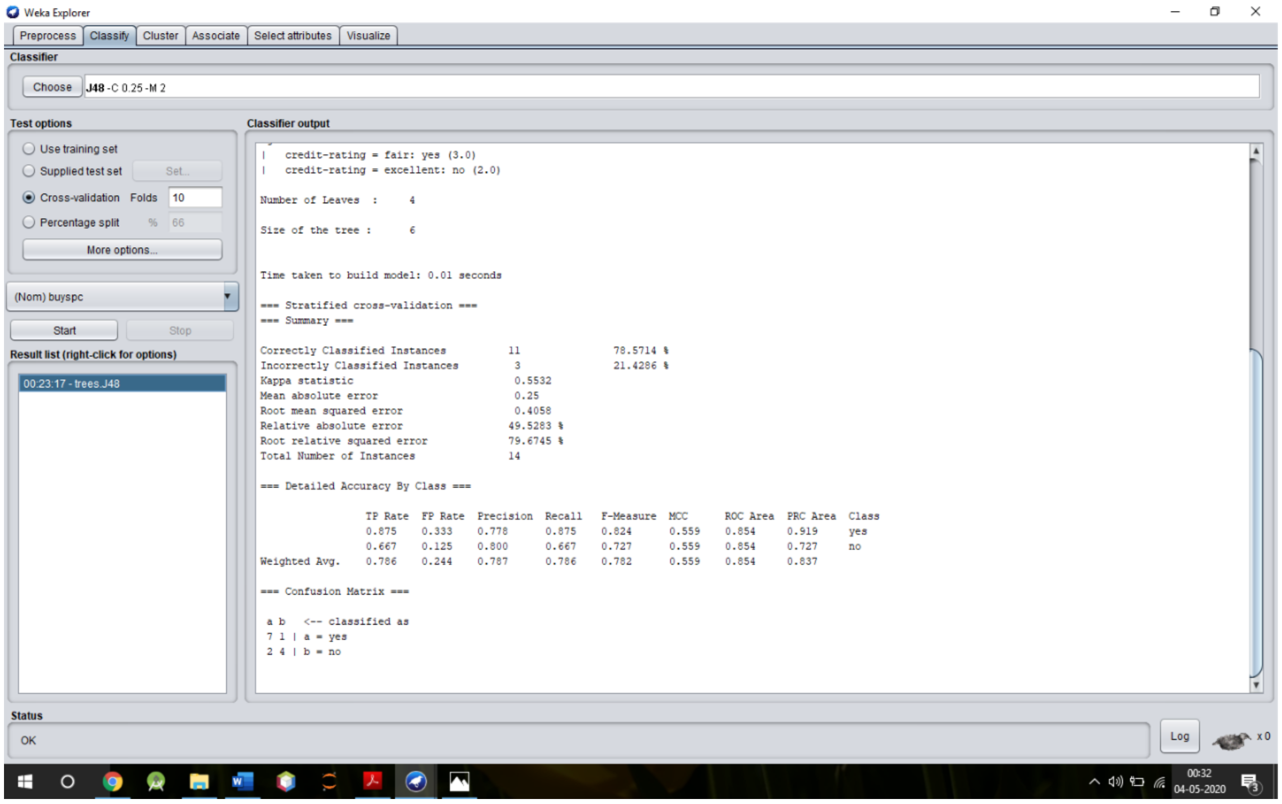
Result list: 00:23:17 - trees.J48

Classifier output

```
=== Run information ===

Scheme:       weka.classifiers.trees.J48 -C 0.25 -M 2
Relation:     student
Instances:    14
Attributes:   5
              age
              income
              student
              credit-rating
              buyspc
Test mode:    10-fold cross-validation

=== Classifier model (full training set) ===

J48 pruned tree
------------------

age = <30: no (5.0/1.0)
age = 30-40: yes (4.0)
age = >40
|   credit-rating = fair: yes (3.0)
|   credit-rating = excellent: no (2.0)

Number of Leaves  :     4

Size of the tree :      6


Time taken to build model: 0.01 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances          11               78.5714 %
Incorrectly Classified Instances         3               21.4286 %
```

Status: OK

00:31
04-05-2020



Classifier: J48 -C 0.25 -M 2

Test options: Cross-validation Folds 10

(Nom) buyspc

Result list: 00:23:17 - trees.J48

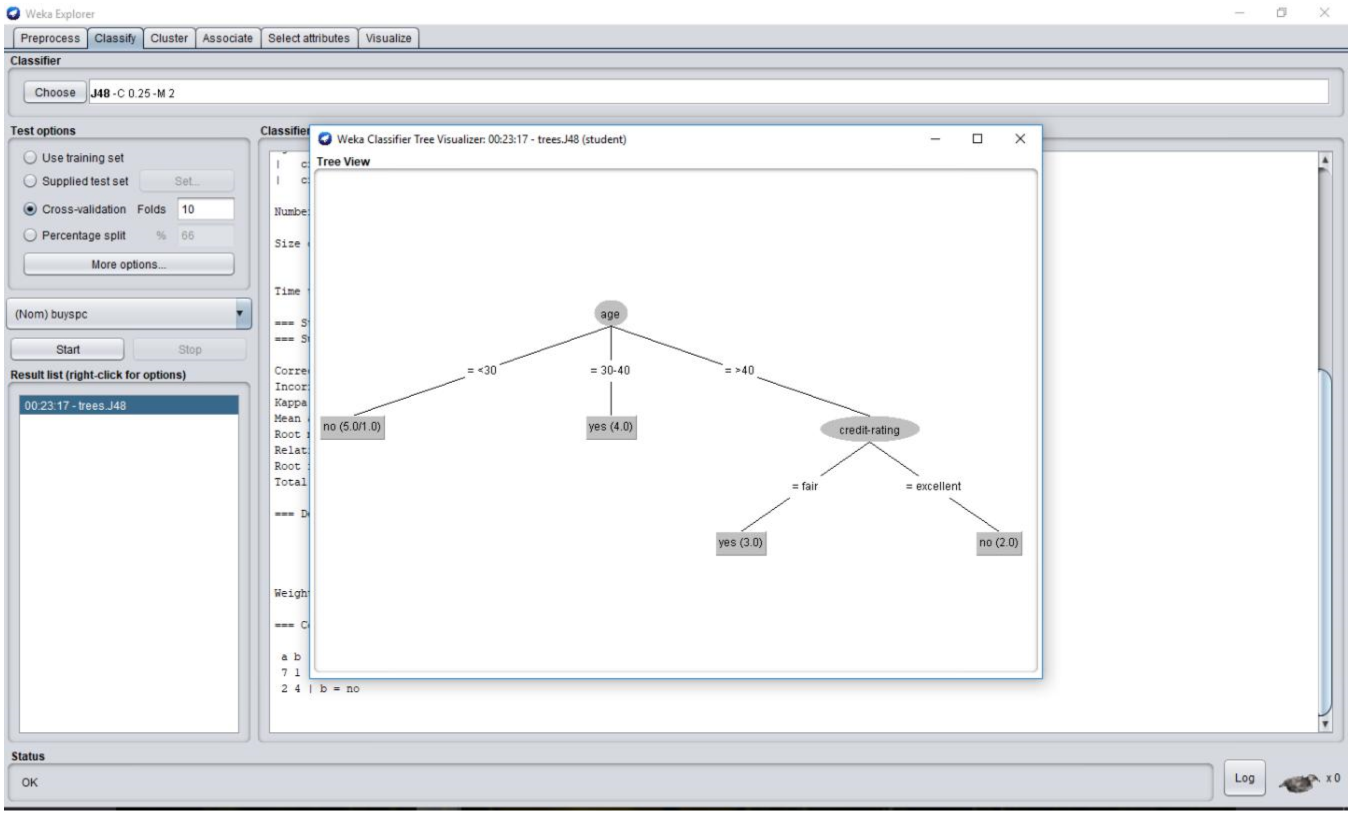
Classifier output

```
|   credit-rating = fair: yes (3.0)
|   credit-rating = excellent: no (2.0)

Number of Leaves  :     4

Size of the tree :      6


Time taken to build model: 0.01 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances          11               78.5714 %
Incorrectly Classified Instances         3               21.4286 %
Kappa statistic                          0.5532
Mean absolute error                      0.25
Root mean squared error                  0.4058
Relative absolute error                 49.5283 %
Root relative squared error             79.6745 %
Total Number of Instances               14

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.875    0.333    0.778      0.875    0.824      0.559    0.854     0.919     yes
                 0.667    0.125    0.800      0.667    0.727      0.559    0.854     0.727     no
Weighted Avg.    0.786    0.244    0.787      0.786    0.782      0.559    0.854     0.837

=== Confusion Matrix ===

 a b   <-- classified as
 7 1 | a = yes
 2 4 | b = no
```

Status: OK

00:32
04-05-2020

Decision Tree



Classifier: J48 -C 0.25 -M 2

Test options: Cross-validation, Folds 10

(Nom) buyspc

Result list: 00:23:17 - trees.J48

Weka Classifier Tree Visualizer: 00:23:17 - trees.J48 (student)

```
 a b
 7 1
 2 4 | b = no
```

Status: OK

Program-6

Aim: Demonstration of classification rule process on Weka data-set using Naïve Bayes algorithm.

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

Dataset :

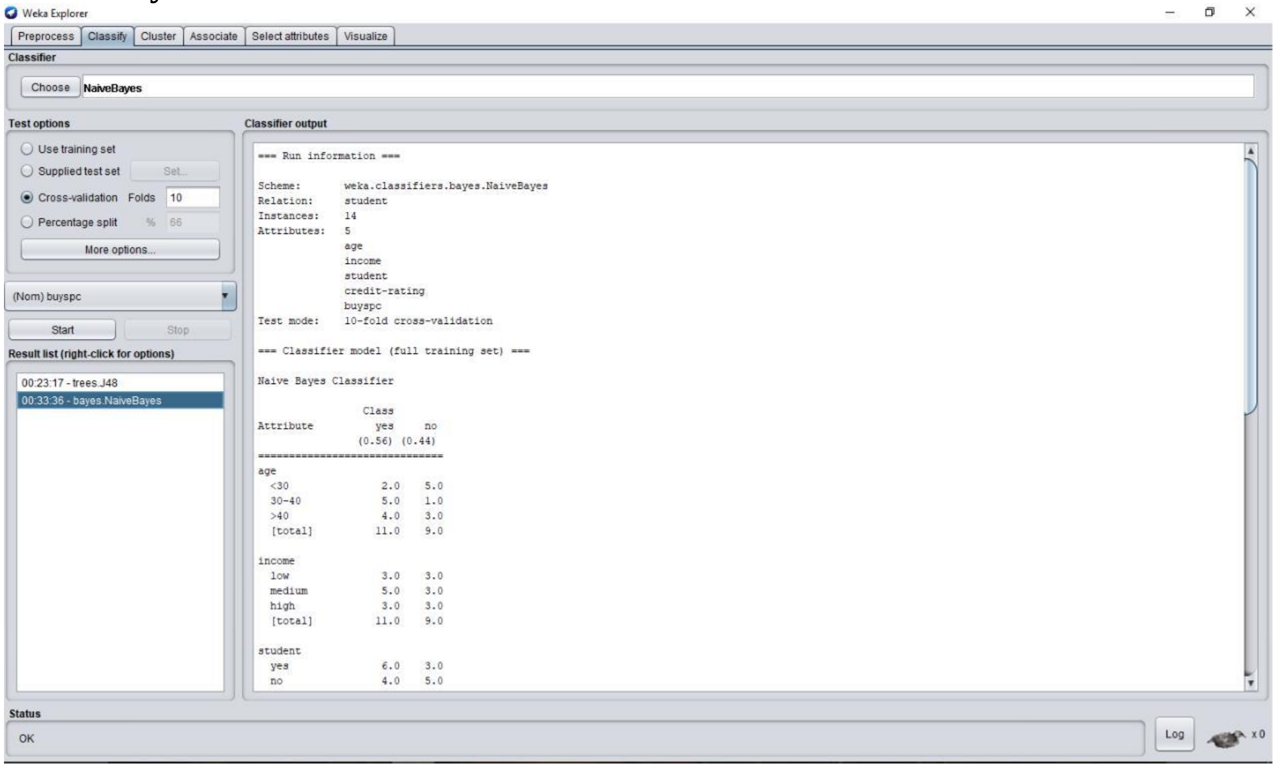
```
@relation student
@attribute age{<30,30-40,>40}
@attribute income {low, medium, high}
@attribute student {yes, no}
@attribute credit-rating {fair, excellent}
 @attribute buyspc {yes, no}
@data
%
<30, high, no, fair, no
<30, high, no, excellent, no
30-40, high, no, fair, yes
>40, medium, no, fair, yes
>40, low, yes, fair, yes
>40, low, yes, excellent, no
30-40, low, yes, excellent, yes
<30, medium, no, fair, no
<30, low, yes, fair, no
>40, medium, yes, fair, yes
<30, medium, yes, excellent, yes
30-40, medium, no, excellent, yes
30-40, high, yes, fair, yes
>40, medium, no, excellent, no
%
```

Steps Involved:

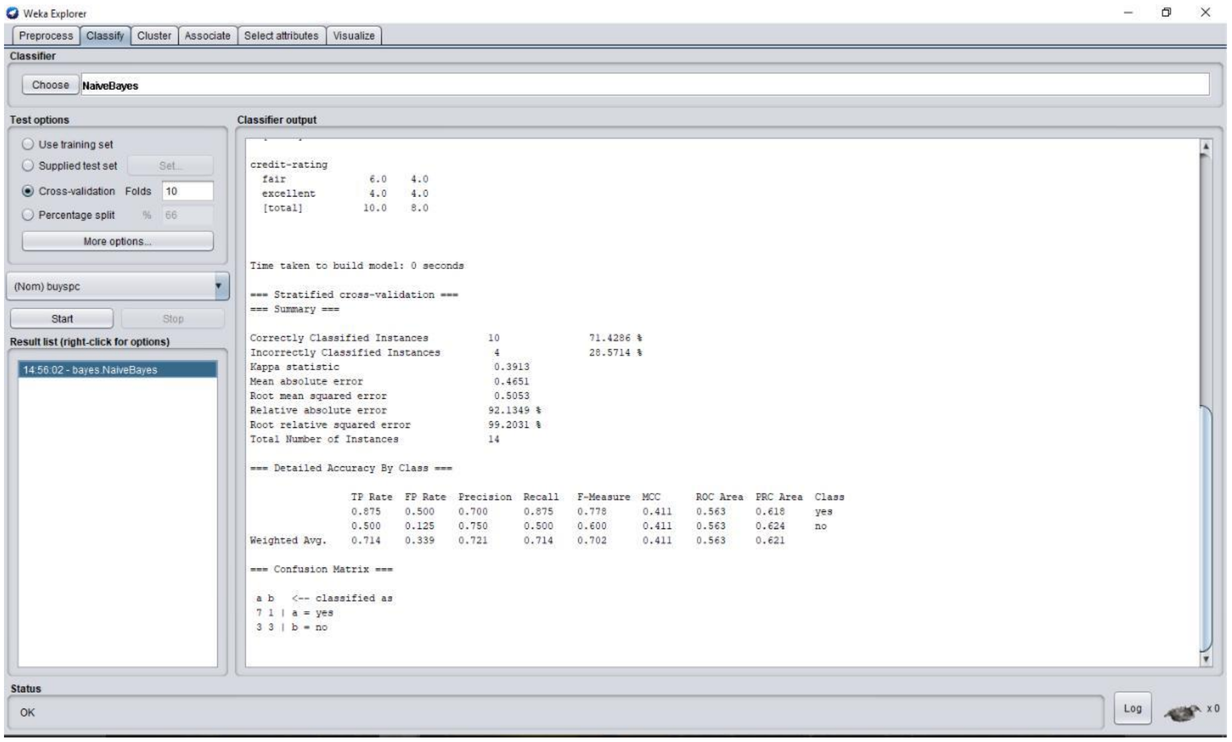
Step 1: Load the data (student.arff) into weka.
Step2: Select the "classify" tab and click "choose" button to select the "NaïveBayes" classifier.
Step3: Specify the various parameters.
Step-4: We now click "start" to generate the model. the ASCII version of the tree as well as evaluation statistic will appear in the right panel when the model construction is complete.
Step-5: Weka also lets us a view a graphical version of the classification tree. This can be done by right clicking the last result set and selecting "visualize tree" from the pop-up menu.

OUTPUT:

NaiveBayes Classifier



```
=== Run information ===

Scheme:       weka.classifiers.bayes.NaiveBayes
Relation:     student
Instances:    14
Attributes:   5
              age
              income
              student
              credit-rating
              buyspc
Test mode:    10-fold cross-validation

=== Classifier model (full training set) ===

Naive Bayes Classifier

                 Class
Attribute          yes     no
                (0.56) (0.44)
==============================
age
  <30              2.0    5.0
  30-40            5.0    1.0
  >40              4.0    3.0
  [total]         11.0    9.0

income
  low              3.0    3.0
  medium           5.0    3.0
  high             3.0    3.0
  [total]         11.0    9.0

student
  yes              6.0    3.0
  no               4.0    5.0
```



```
credit-rating
  fair             6.0    4.0
  excellent        4.0    4.0
  [total]         10.0    8.0



Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances          10               71.4286 %
Incorrectly Classified Instances         4               28.5714 %
Kappa statistic                          0.3913
Mean absolute error                      0.4651
Root mean squared error                  0.5053
Relative absolute error                 92.1349 %
Root relative squared error             99.2031 %
Total Number of Instances               14

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.875    0.500    0.700      0.875    0.778      0.411    0.563     0.618     yes
                 0.500    0.125    0.750      0.500    0.600      0.411    0.563     0.624     no
Weighted Avg.    0.714    0.339    0.721      0.714    0.702      0.411    0.563     0.621

=== Confusion Matrix ===

 a b   <-- classified as
 7 1 | a = yes
 3 3 | b = no
```

Program-7

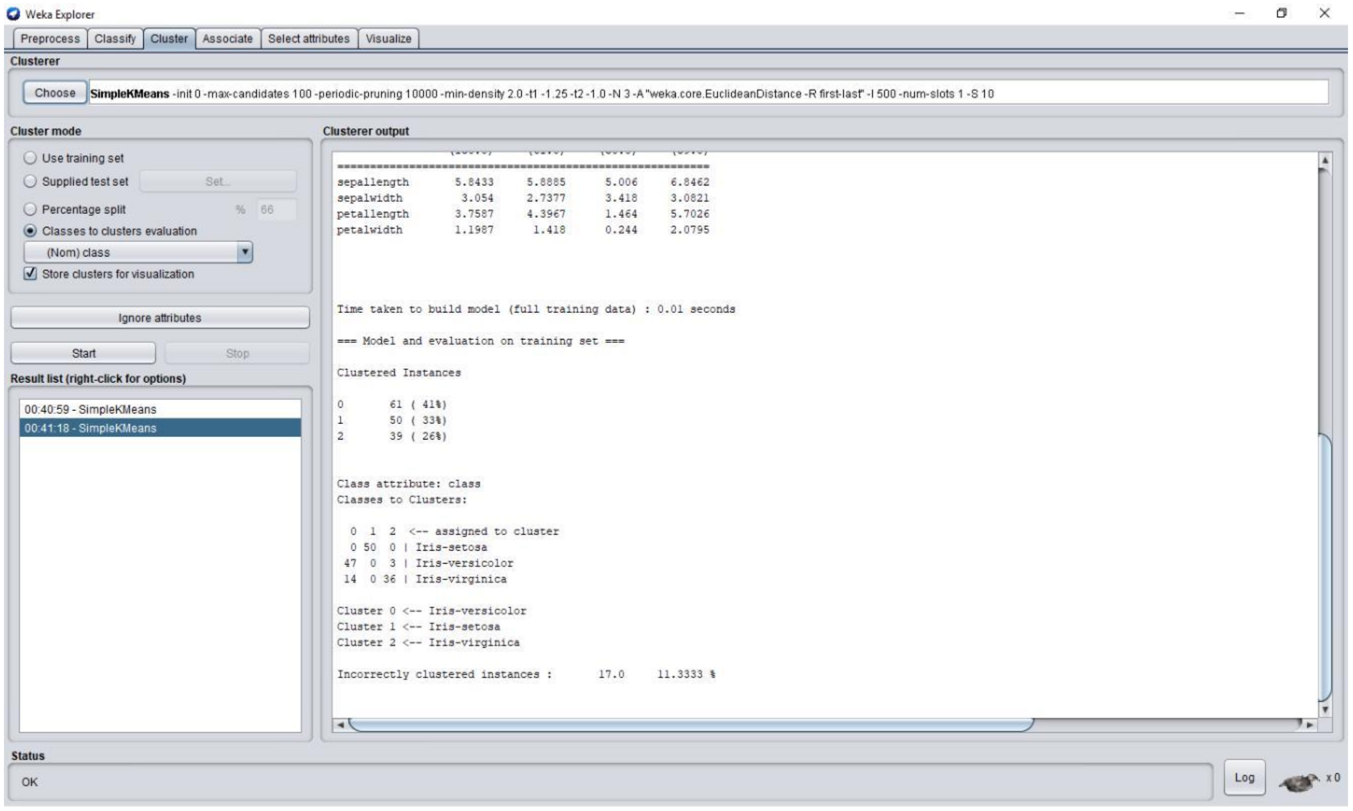
Aim: Demonstration of clustering rule on Weka data-set using simple K-mean.

Theory: K-means clustering is one of the simplest and popular unsupervised machine learning algorithms. A clustering algorithm finds groups of similar instances in the entire dataset. The sample data set used for this example is based on the iris data available in ARFF format. This document assumes that appropriate pre-processing has been performed. This iris dataset includes 150 instances.

Steps Involved:

Step 1: Load the data iris.arff in pre-processing interface
Step2: Select the "cluster" tab and click "choose" button to select the 'simple k-means'
Step3: Specify the various parameters that are number of clusters and value of the seed on as it is. The seed value is used in generating a random number which is used for making the internal assignments of instances of clusters
Step-4: We now click "start" to generate the model. The result window shows the centroid of each cluster as well as statistics on the number and the percent of instances assigned to different clusters. **Step-5:** Select the visualize cluster assignments.

OUTPUT:



Clusterer

SimpleKMeans -init 0 -max-candidates 100 -periodic-pruning 10000 -min-density 2.0 -t1 -1.25 -t2 -1.0 -N 3 -A "weka.core.EuclideanDistance -R first-last" -I 500 -num-slots 1 -S 10

Cluster mode

- Use training set
- Supplied test set
- Percentage split % 66
- [✓ Classes to clusters evaluation]
- (Nom) class
- [✓] Store clusters for visualization

Result list (right-click for options)

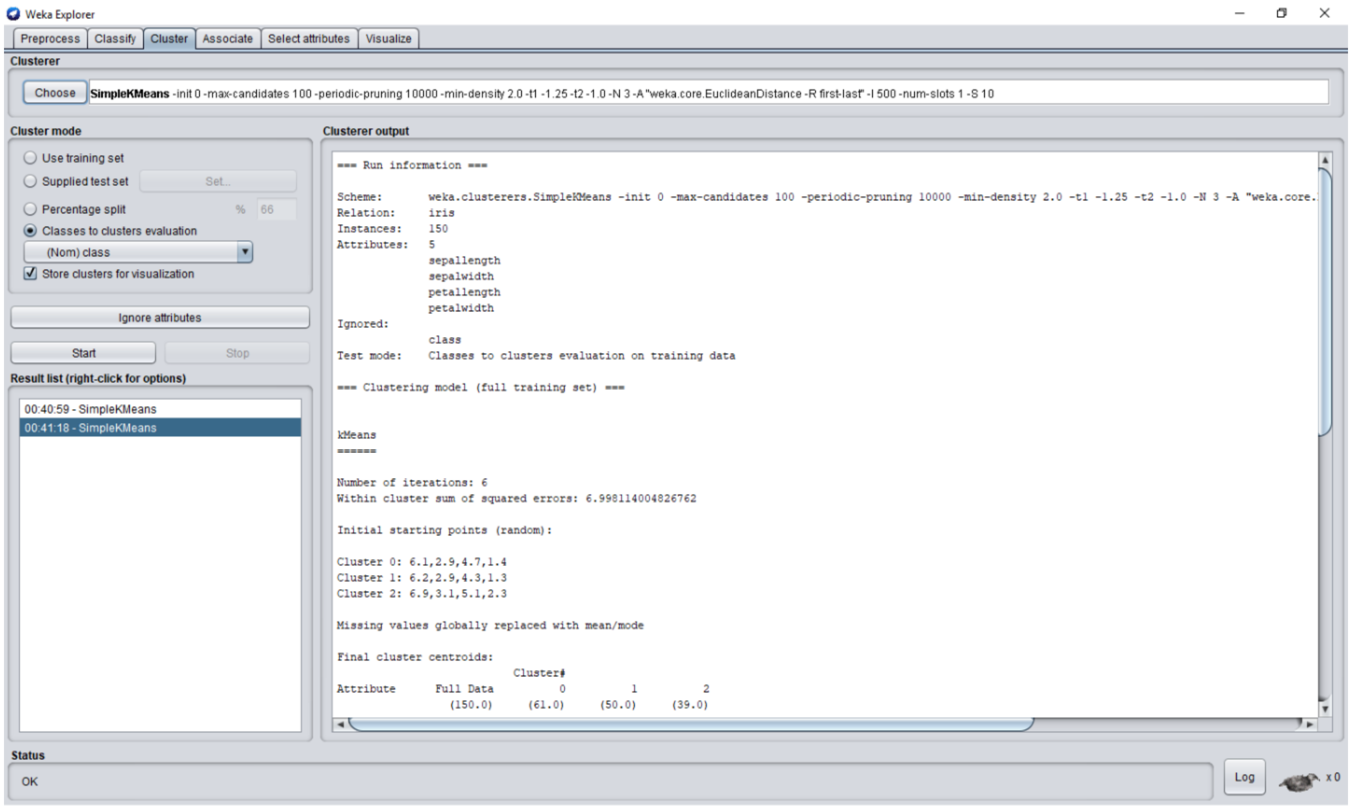
- 00:40:59 - SimpleKMeans
- 00:41:18 - SimpleKMeans

Clusterer output

```
==============================================================
sepallength       5.8433      5.8885       5.006      6.8462
sepalwidth         3.054      2.7377       3.418      3.0821
petallength       3.7587      4.3967       1.464      5.7026
petalwidth        1.1987       1.418       0.244      2.0795




Time taken to build model (full training data) : 0.01 seconds

=== Model and evaluation on training set ===

Clustered Instances

0       61 ( 41%)
1       50 ( 33%)
2       39 ( 26%)


Class attribute: class
Classes to Clusters:

  0  1  2  <-- assigned to cluster
  0 50  0 | Iris-setosa
 47  0  3 | Iris-versicolor
 14  0 36 | Iris-virginica

Cluster 0 <-- Iris-versicolor
Cluster 1 <-- Iris-setosa
Cluster 2 <-- Iris-virginica

Incorrectly clustered instances :       17.0      11.3333 %
```

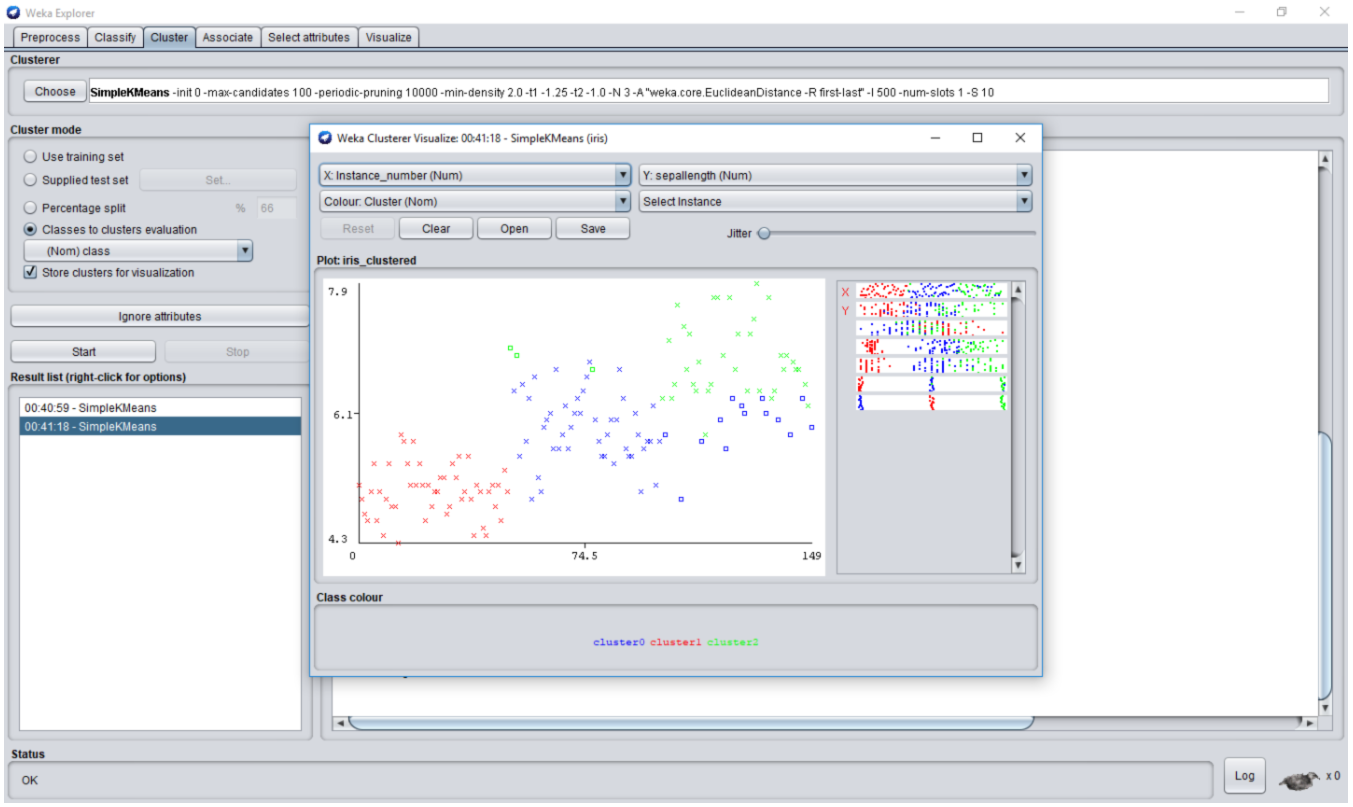
Status

OK



SimpleKMeans -init 0 -max-candidates 100 -periodic-pruning 10000 -min-density 2.0 -t1 -1.25 -t2 -1.0 -N 3 -A "weka.core.EuclideanDistance -R first-last" -I 500 -num-slots 1 -S 10

```
=== Run information ===

Scheme:       weka.clusterers.SimpleKMeans -init 0 -max-candidates 100 -periodic-pruning 10000 -min-density 2.0 -t1 -1.25 -t2 -1.0 -N 3 -A "weka.core.
Relation:     iris
Instances:    150
Attributes:   5
              sepallength
              sepalwidth
              petallength
              petalwidth
Ignored:
              class
Test mode:    Classes to clusters evaluation on training data

=== Clustering model (full training set) ===


kMeans
======

Number of iterations: 6
Within cluster sum of squared errors: 6.998114004826762

Initial starting points (random):

Cluster 0: 6.1,2.9,4.7,1.4
Cluster 1: 6.2,2.9,4.3,1.3
Cluster 2: 6.9,3.1,5.1,2.3

Missing values globally replaced with mean/mode

Final cluster centroids:
                           Cluster#
Attribute      Full Data          0          1          2
                 (150.0)     (61.0)     (50.0)     (39.0)
```



SimpleKMeans -init 0 -max-candidates 100 -periodic-pruning 10000 -min-density 2.0 -t1 -1.25 -t2 -1.0 -N 3 -A "weka.core.EuclideanDistance -R first-last" -I 500 -num-slots 1 -S 10

Program-8

Aim: Write a program of Apriori algorithm using any programming language.

```
#include<iostream.h>
#include<conio.h>
void main()
{
int i,j,t1,k,l,m,f,f1,f2,f3;
//Initial item-purchase
int a[5][5];
for(i=0;i<5;i++)
{
cout<<"\n Enter items from purchase "<<i+1<<":";
for(j=0;j<5;j++)
{
cin>>a[i][j];}
}
//Defining minimum level for acceptence
int min;
cout<<"\n Enter minimum acceptance level";
cin>>min;
//Printing initial input
cout<<"\nInitial Input:\n";
cout<<"\nTrasaction\tItems\n";
for(i=0;i<5;i++)
{
cout<<i+1<<":\t";
for(j=0;j<5;j++)
{
cout<<a[i][j]<<"\t";
}
cout<<"\n";
}
cout<<"\nAssume minimum support: "<<min;
//First pass
int l1[5];
for(i=0;i<5;i++)
{
t1=0;
for(j=0;j<5;j++)
{
for(k=0;k<5;k++)
{
if(a[j][k]==i+1)
{
```

```
t1++;
}
}
}
l1[i]=t1;
}
//Printing first pass
cout<<"\n\nGenerating C1 from data\n";
for(i=0;i<5;i++)
{
cout<<i+1<<": "<<l1[i]<<"\n";
}
//Second pass
//Counting number of possibilities for pass2
int p2pcount=0;
int p2items[5];
int p2pos=0;
for(i=0;i<5;i++)
{
if(l1[i]>=min)
{
p2pcount++;
p2items[p2pos]=i;
p2pos++;
}
}
//Printing selected items for second pass
cout<<"\nGenerating L1 From C1\n";
for(i=0;i<p2pos;i++)
{
cout<<p2items[i]+1<<"\t"<<l1[p2items[i]]<<"\n";
}
//Joining items
int l2[5][3];
int l2t1; //will hold first item for join
int l2t2; //will hold second item for join
int l2pos1=0; //position pointer in l2 array
int l2ocount=0; //product join occruance counter
int l2jcount=0; //join counter
for(i=0;i<p2pcount;i++)
{
for(j=i+1;j<p2pcount;j++)
{
l2t1=p2items[i]+1;
l2t2=p2items[j]+1;
if(l2t1==l2t2)
{
//it is self join
continue;
```

```
}
//join the elements
l2[l2pos1][0]=l2t1;
l2[l2pos1][1]=l2t2;
l2jcount++;
//count occurances
l2ocount=0; //reset counter
for(k=0;k<5;k++)
{
f1=f2=0; //resetting flag
//scan a purcahse
for(l=0;l<5;l++)
{
if(l2t1==a[k][l])
{
//one of the element found
f1=1;
}
if(l2t2==a[k][l])
{
//second elements also found
f2=1;
}
}
//one purchase scanned
if(f1==1&&f2==1) //both items are present in
purchase
{
l2ocount++;
}
}
//assign count
l2[l2pos1][2]=l2ocount;
l2pos1++;
}
}
//Printing second pass
cout<<"\n\nGenerating L2\n";
for(i=0;i<l2jcount;i++)
{
for(j=0;j<3;j++)
{
cout<<l2[i][j]<<"\t";
}
cout<<"\n";
}
//Third pass
int p3pcount=0;
int p3items[5]={-1,-1,-1,-1,-1};
int p3pos=0;
for(i=0;i<5;i++)
{
if(l2[i][2]>=min)
{
f=0;
for(j=0;j<5;j++)
{
if(p3items[j]==l2[i][0])
{
f=1;
}
}
if(f!=1)
{
p3items[p3pos]=l2[i][0];
p3pos++;
p3pcount++;
}
f=0;
for(j=0;j<5;j++)
{
if(p3items[j]==l2[i][1])
{
f=1;
}
}
if(f!=1)
{
p3items[p3pos]=l2[i][1];
p3pos++;
p3pcount++;
}
}
}
//Joining
int l3[5][4];
int l3ocount=0; //occurance counter
int l3jcount=0; //join counter
for(i=0;i<p3pcount;i++)
{
for(j=i+1;j<p3pcount;j++)
{
for(k=j+1;k<p3pcount;k++)
{
l3[i][0]=p3items[i];
l3[i][1]=p3items[j];
l3[i][2]=p3items[k];
l3jcount++;
//count occurances
l3ocount=0; //reset counter
for(k=0;k<5;k++)
{
f1=f2=f3=0; //resetting flag
//scan a purcahse
for(l=0;l<5;l++)
{
if(l3[i][0]==a[k][l])
{
//one of the element found
f1=1;
}
if(l3[i][1]==a[k][l])
{
//second elements also found
f2=1;
}
if(l3[i][2]==a[k][l])
{
//third element also found
f3=1;
}
}
//one purchase scanned
if(f1==1&&f2==1&&f3==1) //all items are present
in purchase
{
l3ocount++;
}
}
//assign count
l3[i][3]=l3ocount;
}
}
}
//Printing second pass
cout<<"\n\nGenerating L3\n";
for(i=0;i<l3jcount;i++)
{
for(j=0;j<4;j++)
{
cout<<l3[i][j]<<"\t";
}
cout<<"\n";
}
getch();
}
```

Output

```
 Enter items from purchase 1:1
5
2
0
0
 Enter items from purchase 2:2
3
4
1
0
 Enter items from purchase 3:3
4
0
0
0
 Enter items from purchase 4:2
1
3
0
0
 Enter items from purchase 5:1
2
3
0
0
 Enter minimum acceptance level3
Initial Input:
Trasaction Items
1: 15200
2: 23410
3: 34000
4: 21300
5: 12300

1: 4
2: 4
3: 4
4: 2
5: 1

1 4
2 4
3 4
Generating L2
1 2 4
1 3 3
2 3 3
```

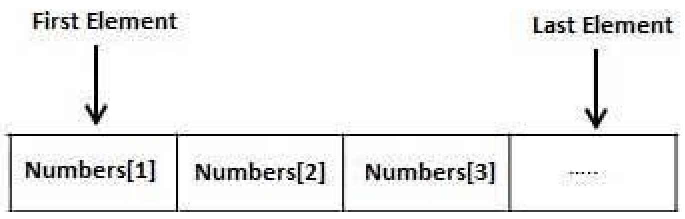
```
Generating L3
1233
*/
```

Practical-9

Aim: Implementation of varying arrays.

Theory:PL/SQL programming language provides a data structure called the VARRAY, which can store a fixed-size sequential collection of elements of the same type. A varray is used to store an ordered collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Procedure:

Creating a Varray Type

A varray type is created with the CREATE TYPE statement. You must specify the maximum size and the type of elements stored in the varray.

The basic syntax for creating a VRRAY type at the schema level is:

```
CREATE OR REPLACE TYPE varray_type_name IS VARRAY(n) of <element_type>
```

Where,

- *varray_type_name* is a valid attribute name,
- *n* is the number of elements (maximum) in the varray,
- *element_type* is the data type of the elements of the array.

Maximum size of a varray can be changed using the ALTER TYPE statement.

For example,

```
CREATE Or REPLACE TYPE namearray AS VARRAY(3) OF VARCHAR2(10);
/

Type created.
```

The basic syntax for creating a VRRAY type within a PL/SQL block is:

```
TYPE varray_type_name IS VARRAY(n) of <element_type>
```

For example:

```
TYPE namearray IS VARRAY(5) OF VARCHAR2(10);
Type grades IS VARRAY(5) OF INTEGER;
```

Example 1

The following program illustrates using varrays:

```
DECLARE
   type namesarray IS VARRAY(5) OF VARCHAR2(10);
   type grades IS VARRAY(5) OF INTEGER; names
   namesarray;
   marks grades;
   total integer;
BEGIN
   names := namesarray('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
   marks:= grades(98, 97, 78, 87, 92); total := names.count;
   dbms_output.put_line('Total '|| total || ' Students');
   FOR i in 1 .. total LOOP
      dbms_output.put_line('Student: ' || names(i) || '
      Marks: ' || marks(i));
   END LOOP;
END;
/
```

Output:

When the above code is executed at SQL prompt, it produces the following result:

```
Student: Kavita Marks: 98
Student: Pritam Marks: 97
Student: Ayan Marks: 78
Student: Rishav Marks: 87
Student: Aziz Marks: 92

PL/SQL procedure successfully completed.
```

Note:

- In oracle environment, the starting index for varrays is always 1.
- You can initialize the varray elements using the constructor method of the varray type, which has the same name as the varray.
- Varrays are one-dimensional arrays.
- A varray is automatically NULL when it is declared and must be initialized before its elements can be referenced.

Practical : 10

Aim: Implementation of Nested Tables.

Theory: A collection is an ordered group of elements having the same data type. Each element is identified by a unique subscript that represents its position in the collection.

PL/SQL provides three collection types:

- Index-by tables or Associative array
- Nested table
- Variable-size array or Varray

Oracle documentation provides the following characteristics for each type of collections:

Collection Type	Number of Elements	Subscript Type	Dense or Sparse	Where Created	Can Be Object Type Attribute
Associative array (or index-by table)	Unbounded	String or integer	Either	Only in PL/SQL block	No
Nested table	Unbounded	Integer	Starts dense, can become sparse	Either in PL/SQL block or at schema level	Yes
Variable-size array (Varray)	Bounded	Integer	Always dense	Either in PL/SQL block or at schema level	Yes

Both types of PL/SQL tables, i.e., index-by tables and nested tables have the same structure and their rows are accessed using the subscript notation. However, these two types of tables differ in one aspect; the nested tables can be stored in a database column and the index-by tables cannot.

Index-By Table

An **index-by** table (also called an associative array) is a set of **key-value** pairs. Each key is unique and is used to locate the corresponding value. The key can be either an integer or a string.
An index-by table is created using the following syntax. Here, we are creating an index-by table named**table_name** whose keys will be of *subscript_type* and associated values will be of *element_type*

```
TYPE type_name IS TABLE OF element_type [NOT NULL] INDEX BY subscript_type;

table_name type_name;
```

Example:

Following example shows how to create a table to store integer values along with names and later it prints the same list of names.

```
DECLARE
   TYPE salary IS TABLE OF NUMBER INDEX BY
   VARCHAR2(20); salary_list salary;
   name   VARCHAR2(20);
 BEGIN
   -- adding elements to the
   table salary_list('Rajnish') :=
   62000;
   salary_list('Minakshi') :=
   75000; salary_list('Martin')
   := 100000;
   salary_list('James') :=
   78000;

   -- printing the table
   name := salary_list.FIRST;
   WHILE name IS NOT null LOOP
      dbms_output.put_line
      ('Salary of ' || name || ' is ' || TO_CHAR(salary_list(name)));
      name := salary_list.NEXT(name);
   END LOOP;
 END;
 /
```

Output:

When the above code is executed at SQL prompt, it produces the following result:

```
Salary of Rajnish is 62000
Salary of Minakshi is 75000
Salary of Martin is 100000
Salary of James is 78000

PL/SQL procedure successfully completed.
```

Example:

Elements of an index-by table could also be a %ROWTYPE of any database table or %TYPE of any database table field. The following example illustrates the concept. We will use the CUSTOMERS table stored in our database as:

```
Select * from customers;

+---- +----------+----- +----------- +----------  +
 | ID | NAME      | AGE | ADDRESS    | SALARY     |
+----   +----------+-----    +-----------+---------          +
|    1 | Ramesh    |   32 | Ahmedabad |   2000.00  |
|    2 | Khilan    |   25 | Delhi     |   1500.00  |
|    3 | kaushik   |   23 | Kota      |   2000.00  |
|    4 | Chaitali  |   25 | Mumbai    |   6500.00  |
|    5 | Hardik    |   27 | Bhopal    |   8500.00  |
|    6 | Komal     |   22 | MP        |   4500.00  |
+----   +-----------+-----    +------------+----------          +
DECLARE
```

```
   CURSOR c_customers is
      select  name from customers;

   TYPE c_list IS TABLE of customers.name%type INDEX BY binary_integer;
   name_list c_list;
   counter integer :=0;
BEGIN
   FOR n IN c_customers LOOP
      counter := counter +1;
      name_list(counter)  := n.name;
      dbms_output.put_line('Customer('||counter||
      '):'||name_list(counter));
  END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal

PL/SQL procedure successfully completed
```

Nested Tables

A **nested table** is like a one-dimensional array with an arbitrary number of elements. However, a nested table differs from an array in the following aspects:

- An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically.
- An array is always dense, i.e., it always has consecutive subscripts. A nested array is dense initially, but it can become sparse when elements are deleted from it.

A **nested table** is created using the following syntax:

```
TYPE type_name IS TABLE OF element_type [NOT NULL];

table_name type_name;
```

This declaration is similar to declaration of an **index-by** table, but there is no INDEX BY clause.

A nested table can be stored in a database column and so it could be used for simplifying SQL operations where you join a single-column table with a larger table. An associative array cannot be stored in the database.

Example:

The following examples illustrate the use of nested table:

```
DECLARE
   TYPE names_table IS TABLE OF
   VARCHAR2(10); TYPE grades IS TABLE OF
   INTEGER;

   names names_table;
   marks grades;
   total integer;
BEGIN
   names := names_table('Kavita', 'Pritam', 'Ayan', 'Rishav',
   'Aziz'); marks:= grades(98, 97, 78, 87, 92); total :=
   names.count;

   dbms_output.put_line('Total '|| total || '
   Students'); FOR i IN 1 .. total LOOP
     dbms_output.put_line('Student:'||names(i)||', Marks:' ||
  marks(i)); end loop;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Total 5 Students
Student:Kavita, Marks:98
Student:Pritam, Marks:97
Student:Ayan, Marks:78
Student:Rishav, Marks:87
Student:Aziz, Marks:92

PL/SQL procedure successfully completed.
```

Example:

Elements of a **nested table** could also be a %ROWTYPE of any database table or %TYPE of any database table field. The following example illustrates the concept. We will use the CUSTOMERS table stored in our database as:

```
Select * from customers;

+----  +----------+-----  +-----------+----------    +
| ID | NAME       | AGE | ADDRESS    | SALARY      |
+---- +----------+----- +-----------+----------    +
|   1 | Ramesh   |   32 | Ahmedabad |   2000.00  |
|   2 | Khilan   |   25 | Delhi     |   1500.00  |
|   3 | kaushik  |   23 | Kota      |   2000.00  |
|   4 | Chaitali |   25 | Mumbai    |   6500.00  |
|   5 | Hardik   |   27 | Bhopal    |   8500.00  |
|   6 | Komal    |   22 | MP        |   4500.00  |
+----  +----------+-----   +-----------+----------        +
DECLARE
   CURSOR c_customers is
      SELECT  name FROM customers;
```

```
   TYPE c_list IS TABLE of
   customers.name%type; name_list c_list :=
   c_list(); counter integer :=0;
BEGIN
   FOR n IN c_customers LOOP
      counter := counter +1;
      name_list.extend;
      name_list(counter)  := n.name;
      dbms_output.put_line('Customer('||counter||'):'||name_list(counter));
   END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal

PL/SQL procedure successfully completed.
```