INDEX

Practical No 1

Aim: Generate Regression model and interpret the result for a given data set.

Theory:

Linear regression is a statistical method used for modeling the relationship between a dependent variable and one or more independent variables by fitting a linear equation to the observed data. It is one of the most fundamental and widely used techniques in statistics and machine learning for predictive modeling and understanding the relationships between variables.

Here are the key components and concepts of linear regression:

1)Dependent Variable (Y): This is the variable you want to predict or explain. It is also known as the response variable or target variable.

2)Independent Variable(s) (X): These are the variables that you believe are related to the dependent variable and are used to predict or explain it. Independent variables are also known as predictors or features.

3)Linear Equation: In simple linear regression, which involves only one independent variable, the relationship between the dependent variable and the independent variable is represented by a linear equation of the form:

Y=b0+b1X+e

- Y is the dependent variable.
- X is the independent variable.
- b0 is the intercept, representing the value of \(Y\) when \(X\) is zero.
- b1 is the slope, representing the change in \(Y\) for a one-unit change in \(X\).

- e represents the error term, accounting for the variability in \(Y\) that cannot be explained by the linear relationship.

4)Regression Coefficients (b0 and b1): These coefficients are estimated from the data using techniques like the least squares method. They represent the strength and direction of the linear relationship between the variables. b0 is the intercept, and b1 is the slope.

5)Residuals (Error Terms): The residuals, denoted by e, are the differences between the observed values of the dependent variable and the values predicted by the linear regression model. These residuals should ideally follow a normal distribution with a mean of zero.

6)Goodness of Fit: Statistical metrics such as the coefficient of determination (R^2) and the mean squared error (MSE) are used to assess how well the linear regression model fits the data. (R^2) measures the proportion of the variance in the dependent variable explained by the model.

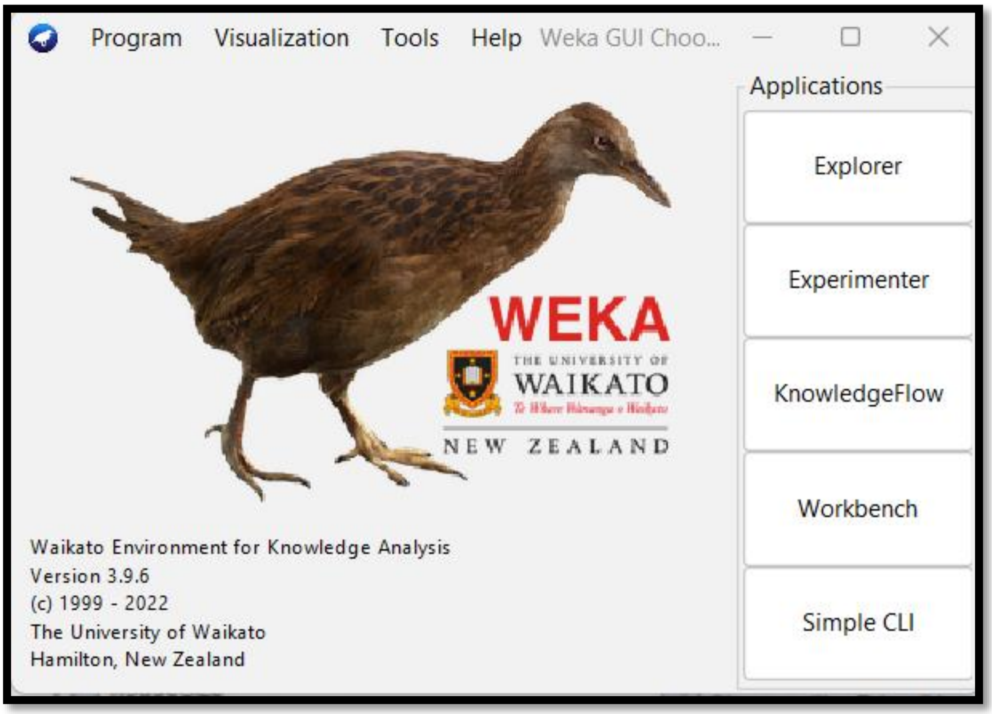
7)Assumptions: Linear regression relies on certain assumptions, including linearity (the relationship is linear), independence of errors (residuals are not correlated), constant variance of errors (homoscedasticity), and normality of residuals.

8)Types of Linear Regression: There are different types of linear regression, including simple linear regression (one independent variable) and multiple linear regression (more than one independent variable). Additionally, variations like polynomial regression and logistic regression adapt the linear regression framework for different types of data and prediction tasks.

Linear regression is widely used in fields such as economics, finance, social sciences, and machine learning for tasks like predicting house prices, modeling stock market trends, understanding the impact of variables on an outcome, and more. It provides valuable insights into the relationships between variables and allows for predictive modeling when the relationship is linear.

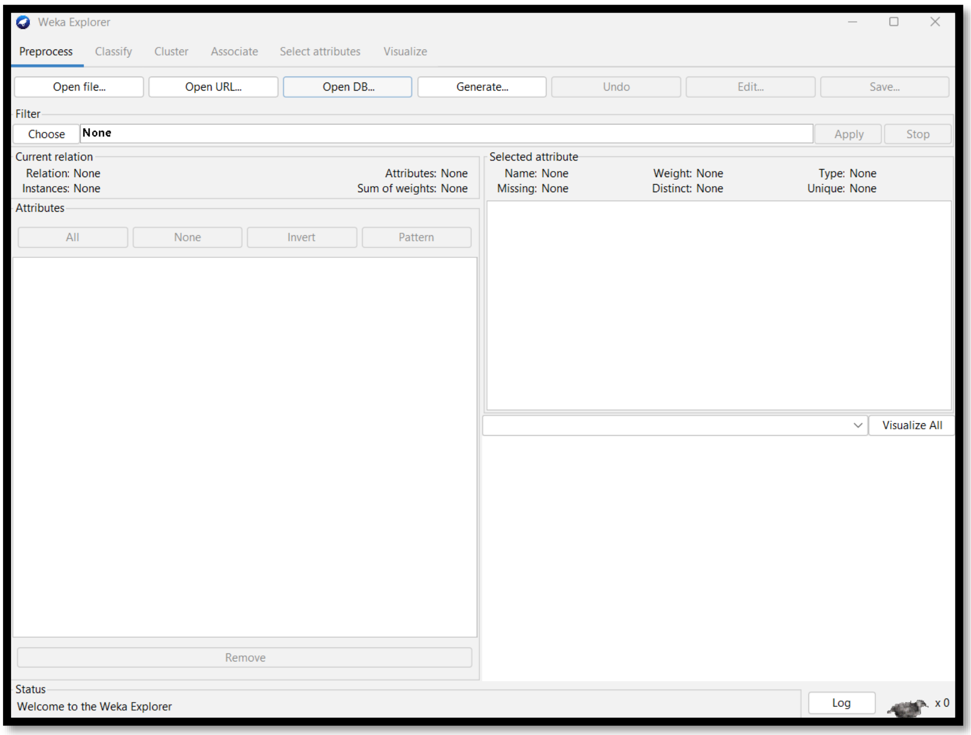
Steps:

1)Open Weka and click on Explorer



Waikato Environment for Knowledge Analysis
Version 3.9.6
(c) 1999 - 2022
The University of Waikato
Hamilton, New Zealand

2) Click on “Open a file”



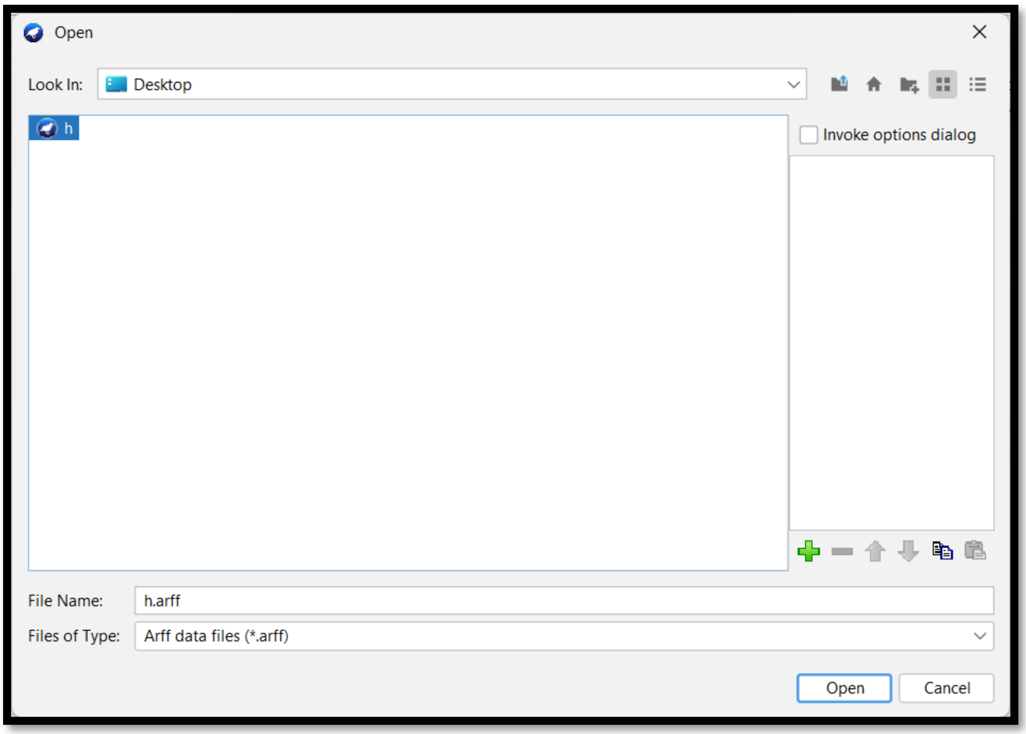
Filter: None

Relation: None
Instances: None
Attributes: None
Sum of weights: None

Name: None
Missing: None
Weight: None
Distinct: None
Type: None
Unique: None

Welcome to the Weka Explorer

3) Open file h.arff in Weka Explorer.



Look In: Desktop

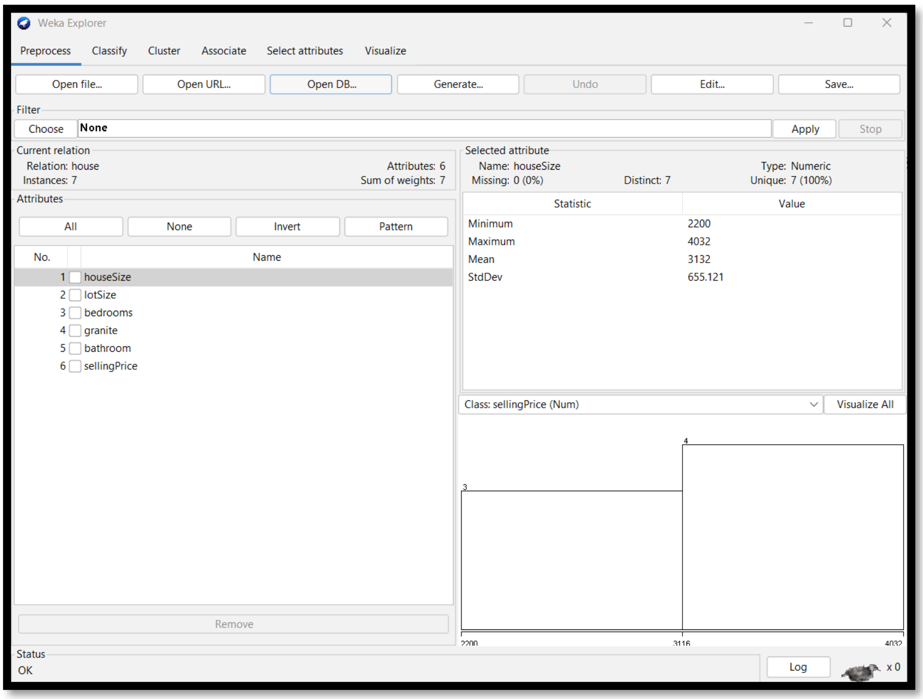
☐ Invoke options dialog

File Name: h.arff

Files of Type: Arff data files (*.arff)

h.arff

```
@RELATION house


@ATTRIBUTE houseSize NUMERIC

@ATTRIBUTE lotSize NUMERIC

@ATTRIBUTE bedrooms NUMERIC

@ATTRIBUTE granite NUMERIC

@ATTRIBUTE bathroom NUMERIC

@ATTRIBUTE sellingPrice NUMERIC


@DATA

3529,9191,6,0,0,205000

3247,10061,5,1,1,224900

4032,10150,5,0,1,197900

2397,14156,4,1,0,189900

2200,9600,4,0,1,195000

3536,19994,6,1,1,325000

2983,9365,5,0,1,230000
```



Filter: None

Current relation
Relation: house
Instances: 7
Attributes: 6
Sum of weights: 7

Attributes

No.	Name
1	□ houseSize
2	□ lotSize
3	□ bedrooms
4	□ granite
5	□ bathroom
6	□ sellingPrice

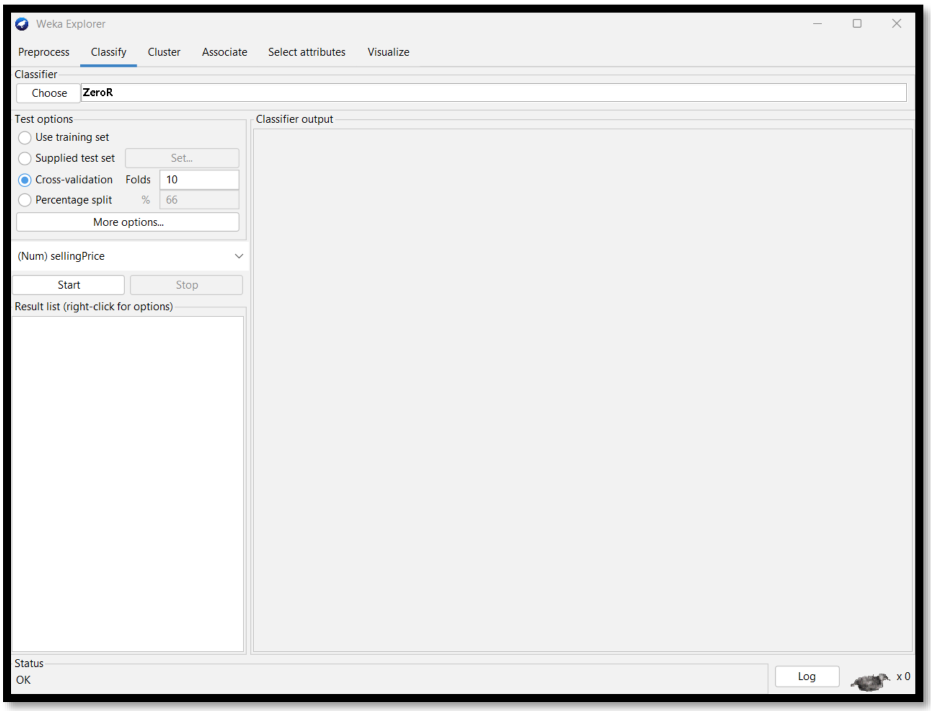
Selected attribute
Name: houseSize
Type: Numeric
Missing: 0 (0%)
Distinct: 7
Unique: 7 (100%)

Statistic	Value
Minimum	2200
Maximum	4032
Mean	3132
StdDev	655.121

Class: sellingPrice (Num)

Status
OK

5)Click on classify



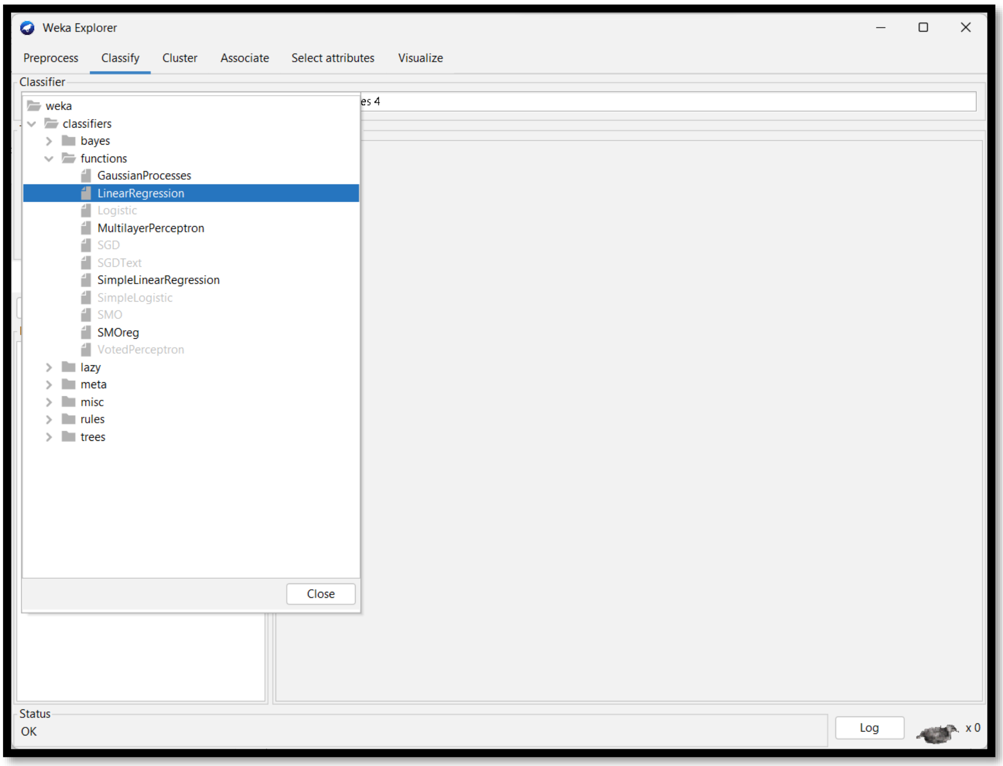
Classifier: ZeroR

○ Use training set
○ Supplied test set
● Cross-validation Folds 10
○ Percentage split % 66

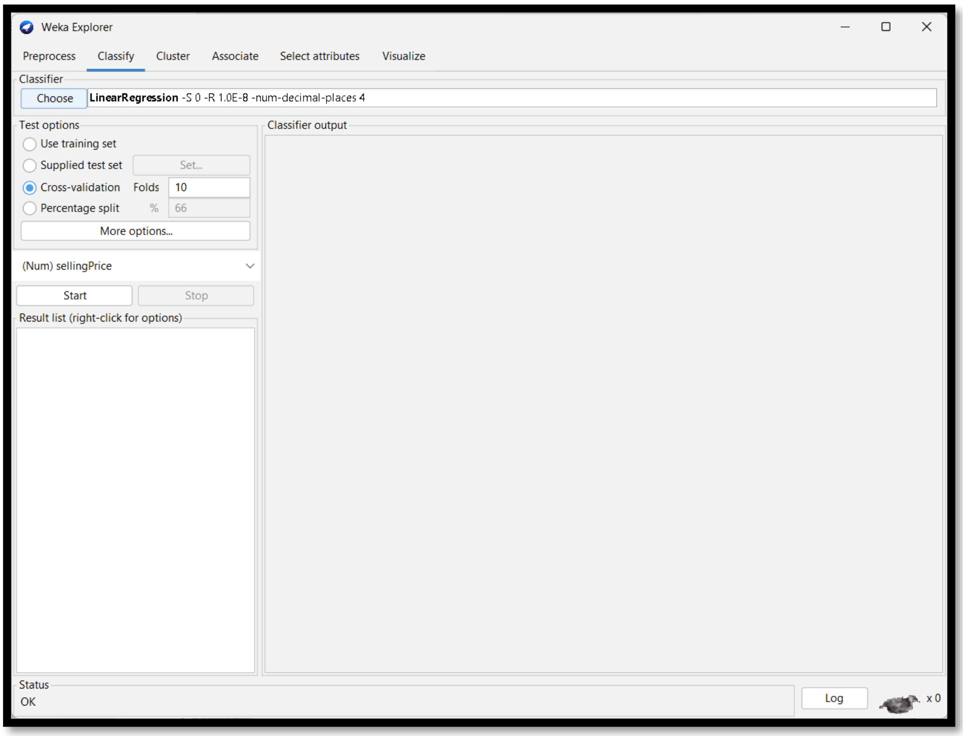
(Num) sellingPrice

Status
OK

6)Choose weka→classifiers→functions→LinearRegression



Status
OK



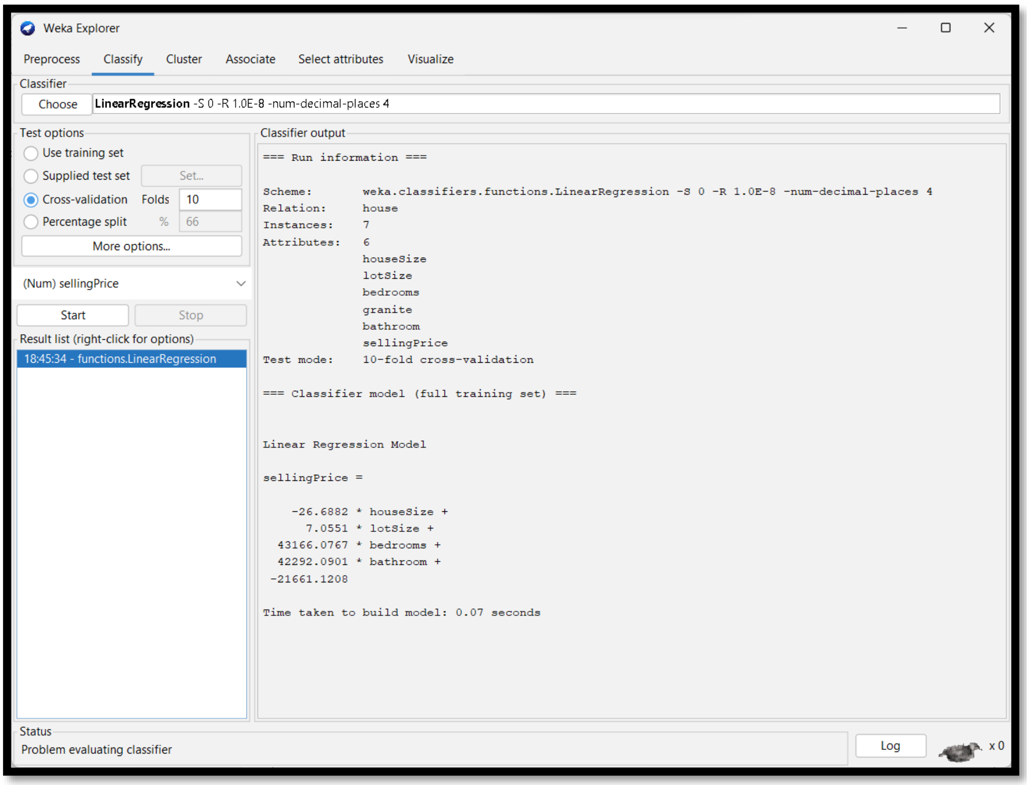
Classifier: LinearRegression -S 0 -R 1.0E-8 -num-decimal-places 4

● Cross-validation Folds 10

(Num) sellingPrice

Status: OK

7)Click on start. You can see the linear regression on the input file.



Classifier: LinearRegression -S 0 -R 1.0E-8 -num-decimal-places 4

● Cross-validation Folds 10

(Num) sellingPrice

18:45:34 - functions.LinearRegression

```
=== Run information ===

Scheme:       weka.classifiers.functions.LinearRegression -S 0 -R 1.0E-8 -num-decimal-places 4
Relation:     house
Instances:    7
Attributes:   6
              houseSize
              lotSize
              bedrooms
              granite
              bathroom
              sellingPrice
Test mode:    10-fold cross-validation

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


Linear Regression Model

sellingPrice =

    -26.6882 * houseSize +
      7.0551 * lotSize +
  43166.0767 * bedrooms +
  42292.0901 * bathroom +
 -21661.1208

Time taken to build model: 0.07 seconds
```

Status: Problem evaluating classifier

Conclusion:

In conclusion, linear regression is a fundamental statistical technique used for modeling and understanding the relationship between a dependent variable and one or more independent variables. It fits a linear equation to observed data, enabling predictions and insights into how changes in independent variables affect the dependent variable. Linear regression's simplicity and interpretability make it a widely applied tool in various fields, from economics to machine learning, for tasks such as predicting outcomes, identifying trends, and quantifying the strength and direction of relationships between variables. However, it assumes certain underlying assumptions, and its effectiveness depends on the linearity of the relationship and the adherence to these assumptions.

Practical No 2

Aim: Generate forecasting model and interpret the result for a given data set.

Theory:

K-means clustering is a simple and widely used unsupervised machine learning algorithm used for clustering data into groups or clusters. Here's a simplified explanation of how it works:

1)Initialization: Start by selecting 'k,' which represents the number of clusters you want to create. Also, randomly initialize 'k' cluster centroids (points that represent the center of each cluster).

2)Assignment: For each data point in your dataset, calculate its distance to each of the 'k' centroids. Assign the data point to the cluster whose centroid is closest to it. This step partitions the data into 'k' clusters.

3)Update Centroids: After assigning all data points to clusters, calculate the mean of all data points within each cluster. This new mean becomes the updated centroid for that cluster.

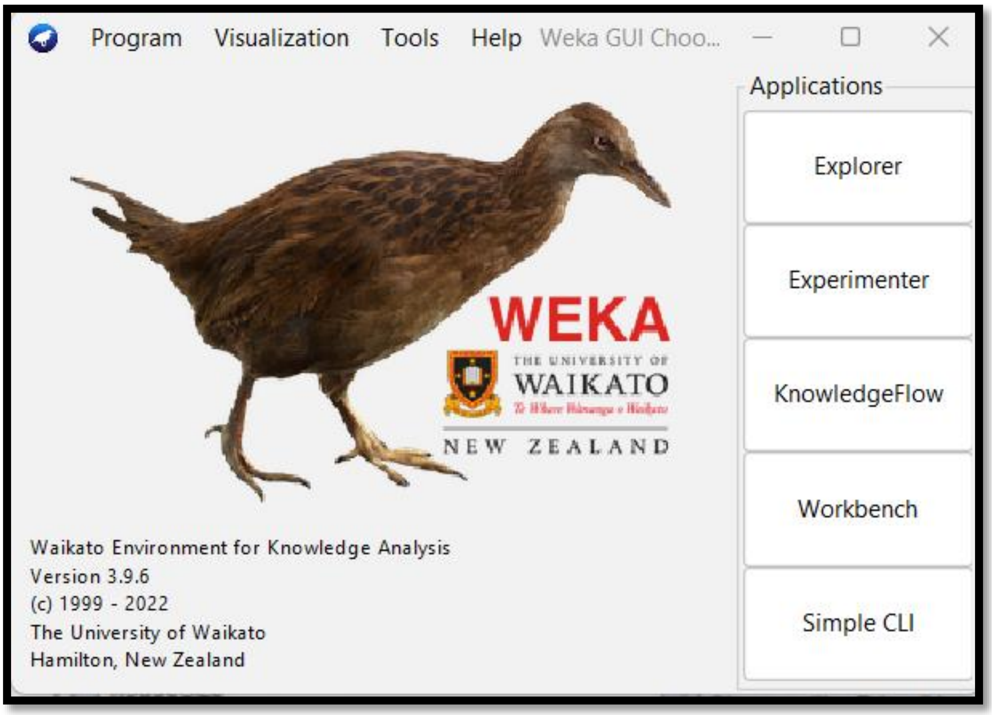
4)Iteration: Repeat the assignment and centroid update steps iteratively until convergence. Convergence occurs when the centroids no longer change significantly or when a predefined number of iterations is reached.

5)Final Result: The algorithm converges to a set of 'k' cluster centroids, and each data point belongs to one of these clusters.

K-means is used for various applications, including image compression, customer segmentation, anomaly detection, and more. However, it has some limitations, such as sensitivity to the initial placement of centroids and difficulties with non-spherical clusters or clusters of different sizes. Researchers have also developed variations and improvements, like K-means++, to address some of these challenges.

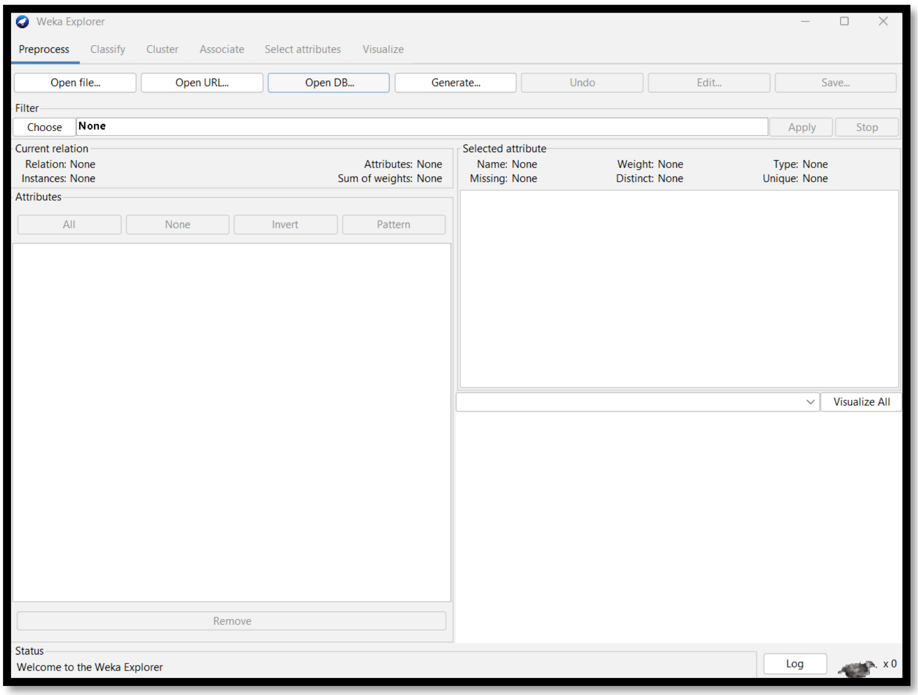
Steps:

1)Open Weka and click on Explorer



Waikato Environment for Knowledge Analysis
Version 3.9.6
(c) 1999 - 2022
The University of Waikato
Hamilton, New Zealand

2) Click on “Open a file”



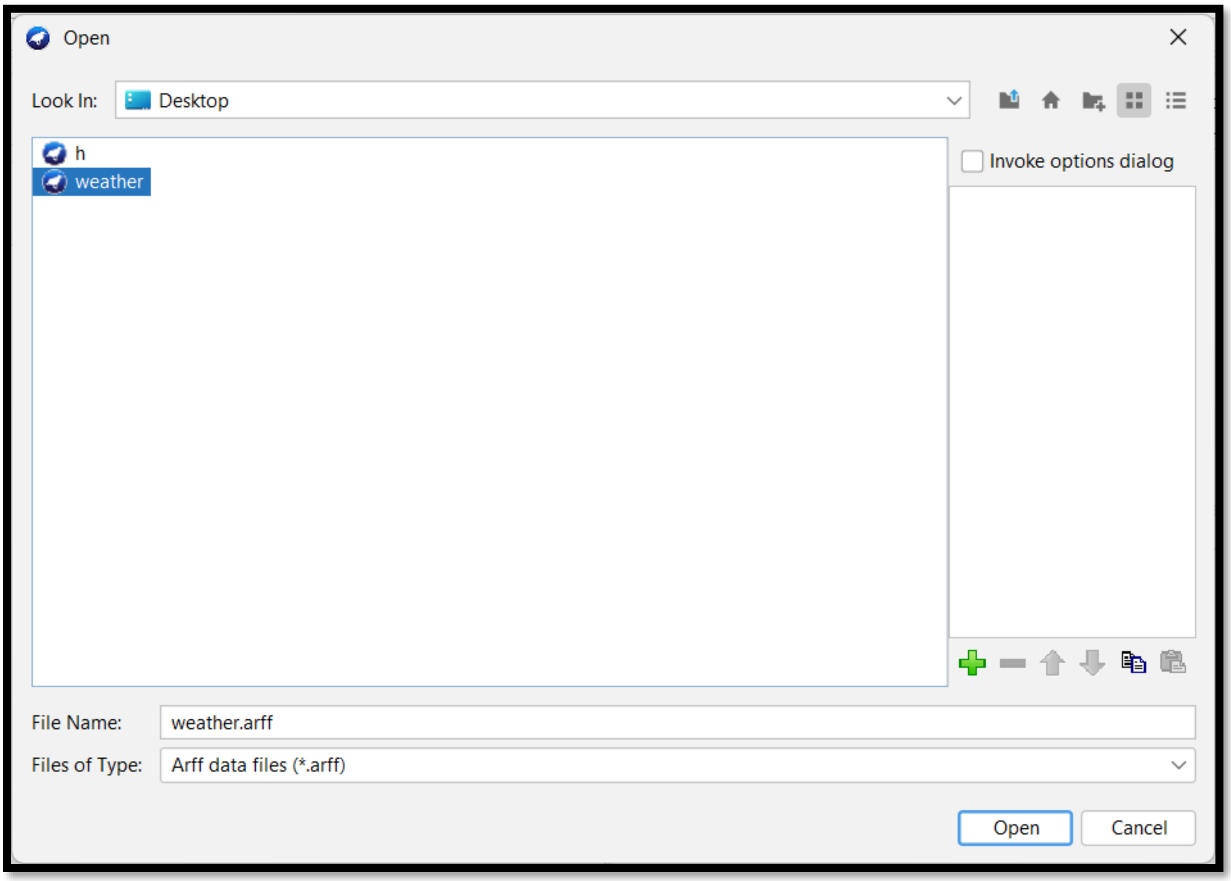
Filter: None

Relation: None
Instances: None
Attributes: None
Sum of weights: None

Name: None
Missing: None
Weight: None
Distinct: None
Type: None
Unique: None

Welcome to the Weka Explorer

3) Open file weather.arff in Weka Explorer.



Look In: Desktop

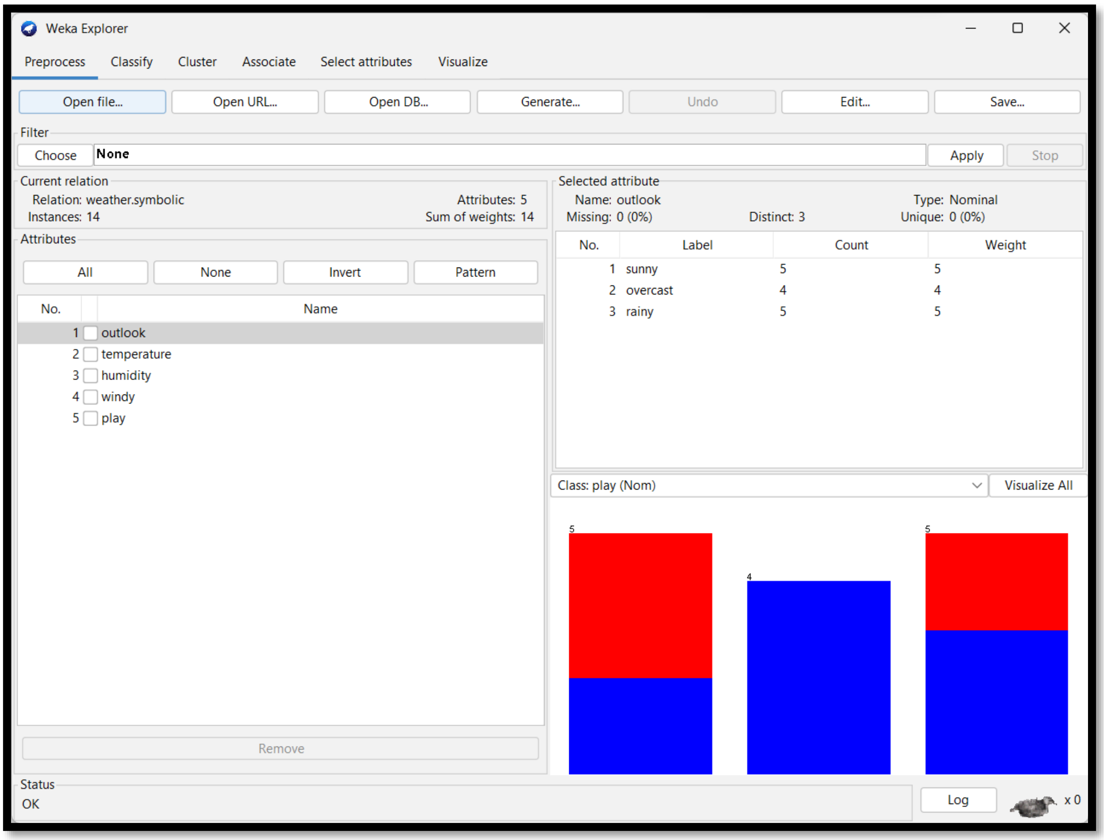
□ Invoke options dialog

File Name: weather.arff

Files of Type: Arff data files (*.arff)

weather.arff

```
@relation weather.symbolic

@attribute outlook {sunny, overcast, rainy}
@attribute temperature {hot, mild, cool}
@attribute humidity {high, normal}
@attribute windy {TRUE, FALSE}
@attribute play {yes, no}

@data
sunny,hot,high,FALSE,no
sunny,hot,high,TRUE,no
overcast,hot,high,FALSE,yes
rainy,mild,high,FALSE,yes
rainy,cool,normal,FALSE,yes
rainy,cool,normal,TRUE,no
overcast,cool,normal,TRUE,yes
sunny,mild,high,FALSE,no
sunny,cool,normal,FALSE,yes
rainy,mild,normal,FALSE,yes
sunny,mild,normal,TRUE,yes
overcast,mild,high,TRUE,yes
overcast,hot,normal,FALSE,yes
rainy,mild,high,TRUE,no
```



Filter

Choose None

Current relation

Relation: weather.symbolic Attributes: 5

Instances: 14 Sum of weights: 14

Attributes

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

Selected attribute

Name: outlook Type: Nominal

Missing: 0 (0%) Distinct: 3 Unique: 0 (0%)

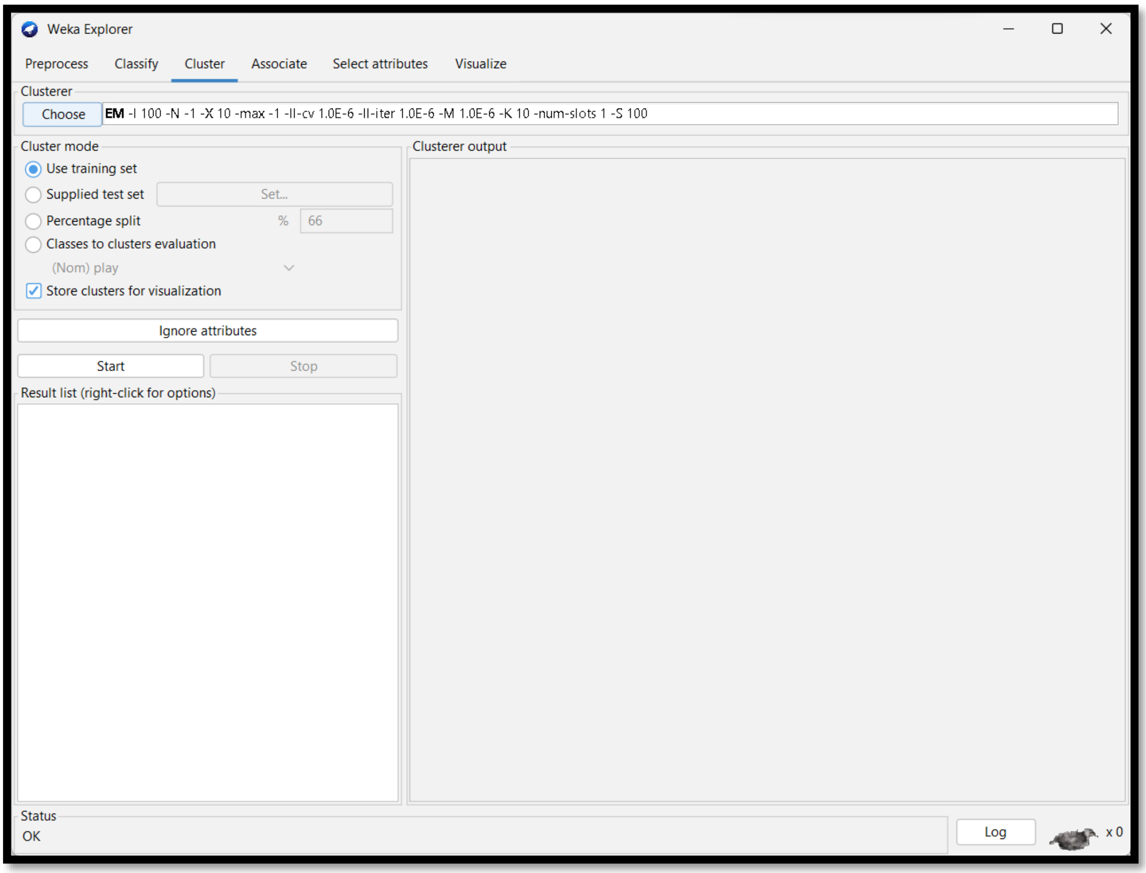
No.	Label	Count	Weight
1	sunny	5	5
2	overcast	4	4
3	rainy	5	5

Class: play (Nom)

Status

OK

5)Click on cluster



Clusterer

Choose EM -I 100 -N -1 -X 10 -max -1 -ll-cv 1.0E-6 -ll-iter 1.0E-6 -M 1.0E-6 -K 10 -num-slots 1 -S 100

Cluster mode

● Use training set

○ Supplied test set

○ Percentage split % 66

○ Classes to clusters evaluation

(Nom) play

☑ Store clusters for visualization

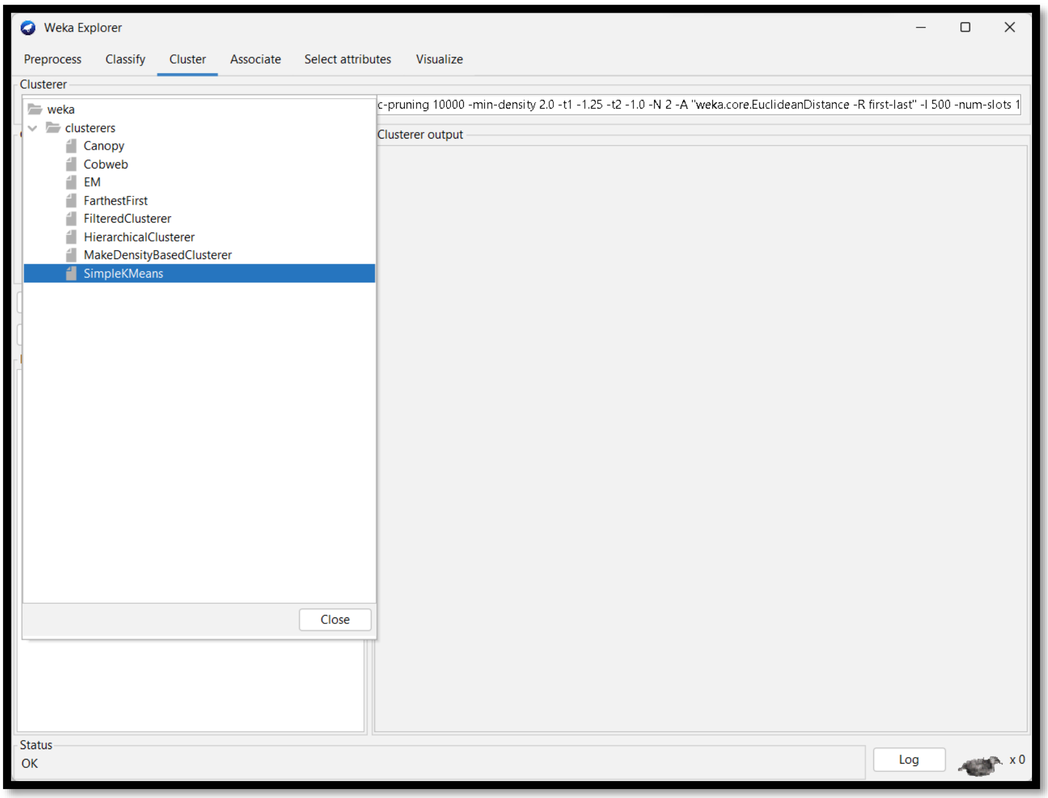
Result list (right-click for options)

Clusterer output

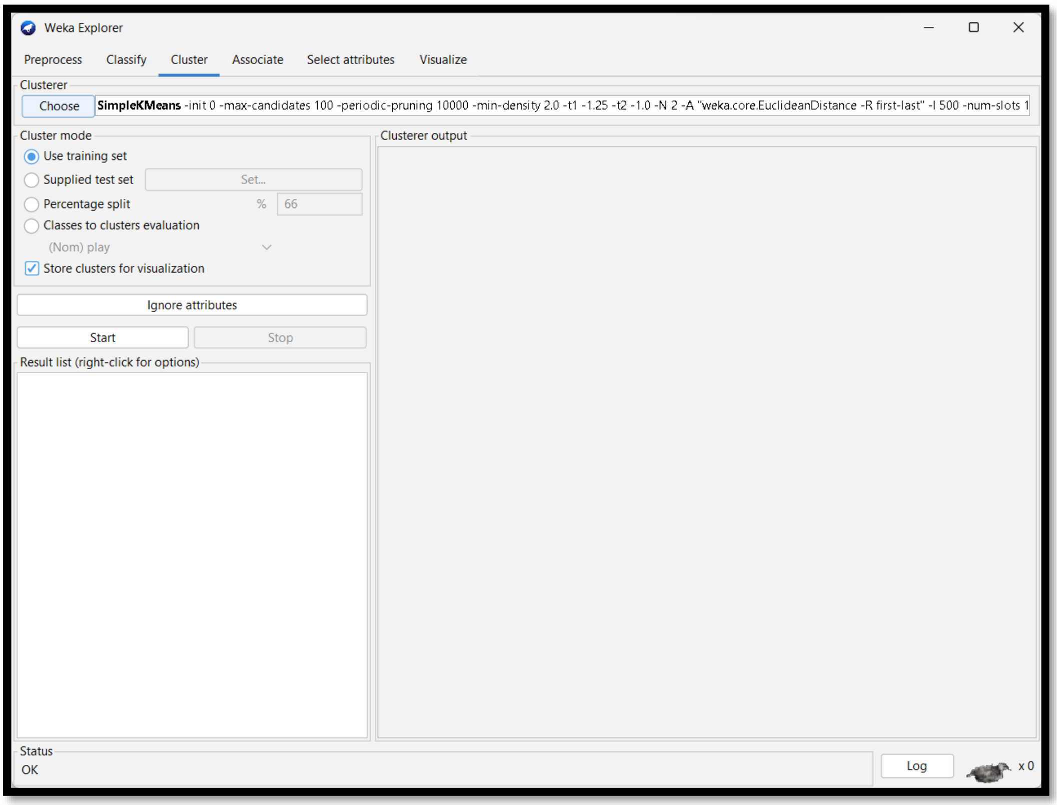
Status

OK

6)Choose weka→clusters→SimpleKMeans



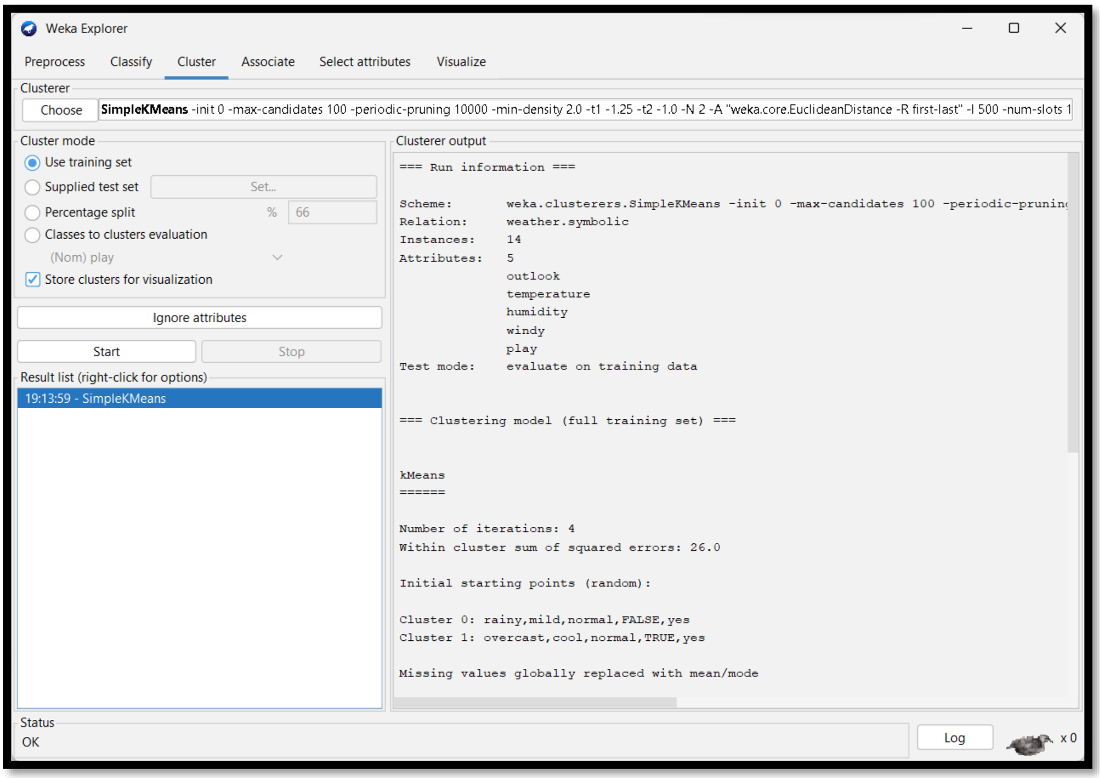
```
c-pruning 10000 -min-density 2.0 -t1 -1.25 -t2 -1.0 -N 2 -A "weka.core.EuclideanDistance -R first-last" -I 500 -num-slots 1
```



```
SimpleKMeans -init 0 -max-candidates 100 -periodic-pruning 10000 -min-density 2.0 -t1 -1.25 -t2 -1.0 -N 2 -A "weka.core.EuclideanDistance -R first-last" -I 500 -num-slots 1
```

- ● Use training set
- ○ Supplied test set
- ○ Percentage split % 66
- ○ Classes to clusters evaluation
- (Nom) play
- ☑ Store clusters for visualization

7)Click on start. You can see the linear regression on the input file.



Clusterer: SimpleKMeans -init 0 -max-candidates 100 -periodic-pruning 10000 -min-density 2.0 -t1 -1.25 -t2 -1.0 -N 2 -A "weka.core.EuclideanDistance -R first-last" -I 500 -num-slots 1

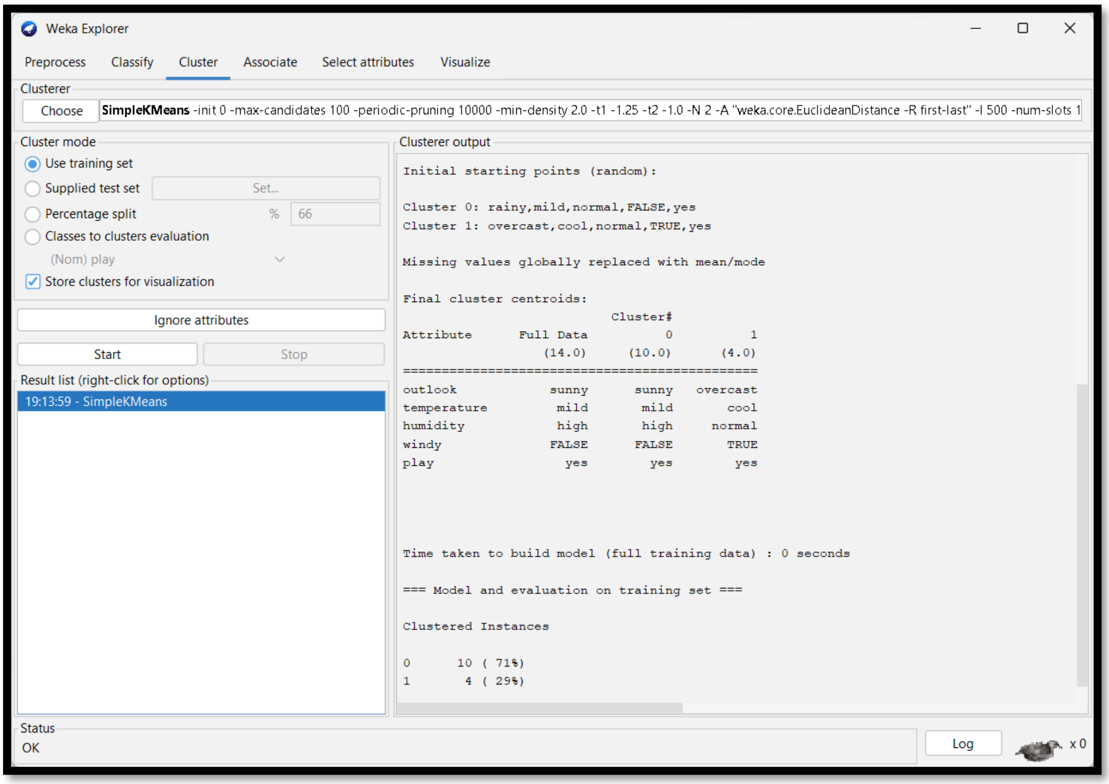
Cluster mode
- ● Use training set
- ○ Supplied test set
- ○ Percentage split % 66
- ○ Classes to clusters evaluation (Nom) play
- ☑ Store clusters for visualization

Result list (right-click for options)
19:13:59 - SimpleKMeans

Status
OK

Clusterer output

```
=== Run information ===

Scheme:       weka.clusterers.SimpleKMeans -init 0 -max-candidates 100 -periodic-prunin
Relation:     weather.symbolic
Instances:    14
Attributes:   5
              outlook
              temperature
              humidity
              windy
              play
Test mode:    evaluate on training data


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


kMeans
======

Number of iterations: 4
Within cluster sum of squared errors: 26.0

Initial starting points (random):

Cluster 0: rainy,mild,normal,FALSE,yes
Cluster 1: overcast,cool,normal,TRUE,yes

Missing values globally replaced with mean/mode
```



Clusterer: SimpleKMeans -init 0 -max-candidates 100 -periodic-pruning 10000 -min-density 2.0 -t1 -1.25 -t2 -1.0 -N 2 -A "weka.core.EuclideanDistance -R first-last" -I 500 -num-slots 1

Cluster mode
- ● Use training set
- ○ Supplied test set
- ○ Percentage split % 66
- ○ Classes to clusters evaluation (Nom) play
- ☑ Store clusters for visualization

Result list (right-click for options)
19:13:59 - SimpleKMeans

Status
OK

Clusterer output

```
Initial starting points (random):

Cluster 0: rainy,mild,normal,FALSE,yes
Cluster 1: overcast,cool,normal,TRUE,yes

Missing values globally replaced with mean/mode

Final cluster centroids:
                           Cluster#
Attribute      Full Data          0          1
                  (14.0)     (10.0)      (4.0)
===============================================
outlook            sunny      sunny   overcast
temperature         mild       mild       cool
humidity            high       high     normal
windy              FALSE      FALSE       TRUE
play                 yes        yes        yes




Time taken to build model (full training data) : 0 seconds

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

Clustered Instances

0       10 ( 71%)
1        4 ( 29%)
```

Conclusion:

In conclusion, K-means clustering is a widely used unsupervised machine learning technique that partitions data into 'k' clusters based on similarity. It's a simple yet effective method for grouping data points, finding patterns, and discovering underlying structures within a dataset. K-means is valuable in various domains, including customer segmentation, image compression, and anomaly detection. However, its performance can be affected by the initial placement of centroids and its sensitivity to data distribution. Careful selection of 'k' and consideration of data preprocessing are essential for achieving meaningful and interpretable results with K-means clustering.

Practical No 3

Aim: Write a map-reduce program to count the number of occurrences of each word in the given dataset.

Theory:

MapReduce is a programming model and data processing framework developed by Google and popularized by Apache Hadoop. It's designed for processing and generating large datasets that can be distributed across a cluster of computers. MapReduce simplifies the process of writing distributed and parallelized data processing applications. It consists of two main phases: the "Map" phase and the "Reduce" phase.

1. Map Phase:

- **Mapping:** In this phase, the input data is divided into chunks and processed in parallel. Each chunk is processed by a "mapper" function that extracts and emits key-value pairs based on the data's characteristics. The mapper function takes the input data, processes it, and emits intermediate key-value pairs.
- **Shuffling and Sorting:** After mapping, all emitted key-value pairs are shuffled and sorted by key. This process groups together all values associated with the same key. This prepares the data for the next phase.

2. Reduce Phase:

- **Reducing:** In this phase, a "reducer" function takes the sorted and grouped key-value pairs from the shuffle and sort phase. It then processes these key-value pairs to produce the final output. The reducer function can perform operations like aggregation, summarization, or any other computations based on the keys.

- **Output:** The final results are typically written to an output file or storage system.

Here are some key characteristics of MapReduce:

- Parallel Processing: MapReduce is designed to work in a distributed and parallelized manner. It processes data across multiple nodes in a cluster, which can significantly speed up data processing.

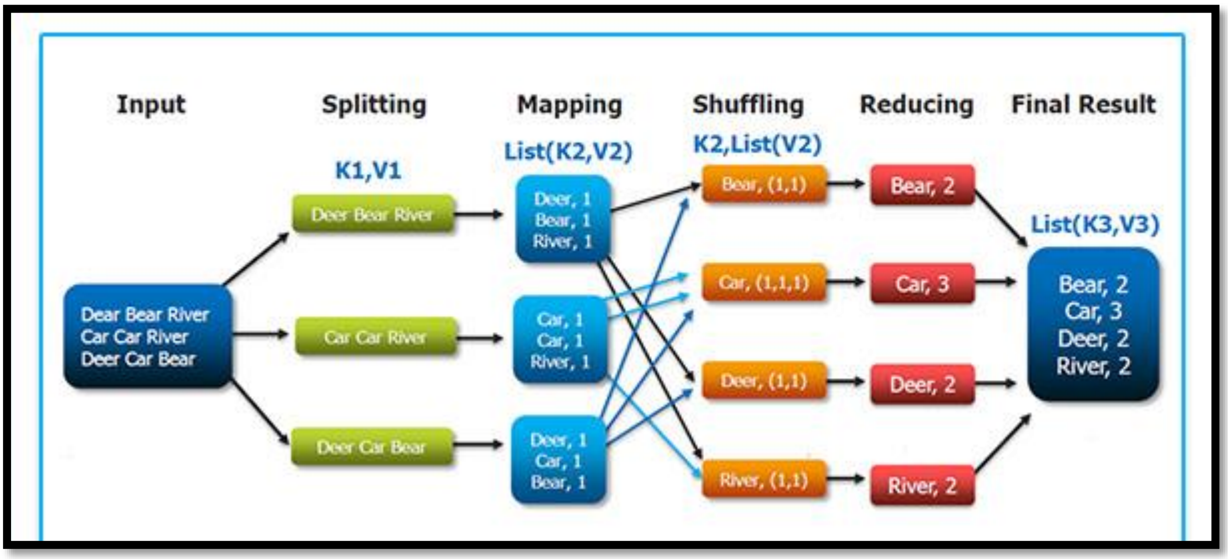
- Fault Tolerance: MapReduce frameworks like Hadoop provide fault tolerance. If a node in the cluster fails during processing, the framework can automatically reroute the work to other nodes, ensuring that the job continues to run.

- Scalability: MapReduce scales well with large datasets. You can add more nodes to your cluster to handle larger volumes of data.

- Simplicity: MapReduce abstracts many of the complexities of distributed processing, allowing developers to focus on writing the mapping and reducing functions without worrying about low-level distributed computing details.

MapReduce has been widely used for processing vast amounts of data, especially in big data and distributed computing environments. It's a foundational concept in many big data ecosystems, including Hadoop, and has been the basis for a wide range of data processing tasks, including log analysis, data transformation, and machine learning.

Word Count Example:



Code:

WordCount.java(Driver Class)

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount extends Configured implements Tool {
  public int run(String[] args) throws Exception {
    // creating a JobConf object and assigning a job name for identification purposes
    JobConf conf = new JobConf(getConf(), WordCount.class);
    conf.setJobName("WordCount");

    // Setting configuration object with the Data Type of output Key and Value
    conf.setOutputKeyClass(Text.class);
```

```
        conf.setOutputValueClass(IntWritable.class);

        // Providing the mapper and reducer class names
        conf.setMapperClass(WordCountMapper.class);
        conf.setReducerClass(WordCountReducer.class);

        // We will give 2 arguments at the run time, one for the input path and the other for the output path
        Path inp = new Path(args[0]);
        Path out = new Path(args[1]);

        // the HDFS input and output directory to be fetched from the command line
        FileInputFormat.addInputPath(conf, inp);
        FileOutputFormat.setOutputPath(conf, out);
        JobClient.runJob(conf);

        return 0;
    }

    public static void main(String[] args) throws Exception {
        // this main function will call the run method defined above.
        int res = ToolRunner.run(new Configuration(), new WordCount(), args);
        System.exit(res);
    }
}
```

WordCountMapper.java(Mapper Class)

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
```

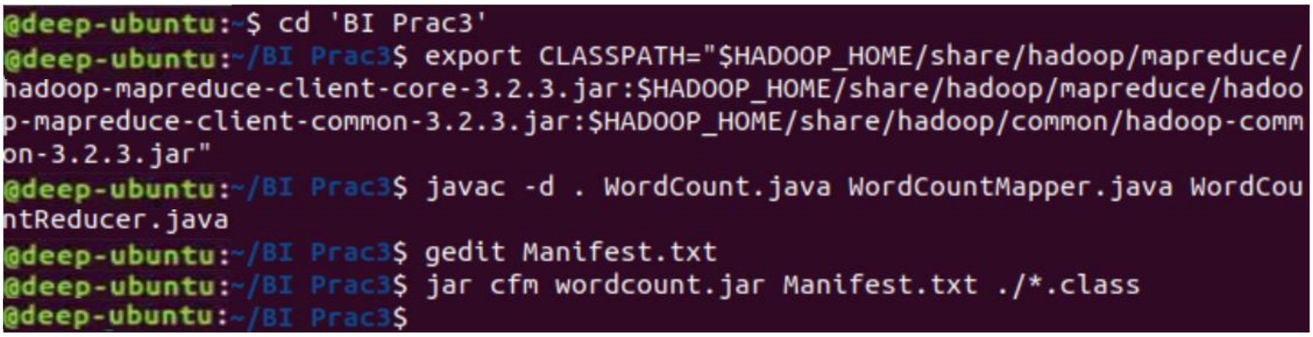
```
public class WordCountMapper extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable> {

    // Hadoop supported data types
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    // Map method that performs the tokenizer job and framing the initial key-value pairs
    // After all lines are converted into key-value pairs, the reducer is called.
    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable>
output, Reporter reporter) throws IOException {
        // Taking one line at a time from the input file and tokenizing the same
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);

        // Iterating through all the words available in that line and forming the key-value pair
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            // Sending to the output collector, which in turn passes the same to the reducer
            output.collect(word, one);
        }
    }
}
```

WordCountReducer.java(Reducer Class)

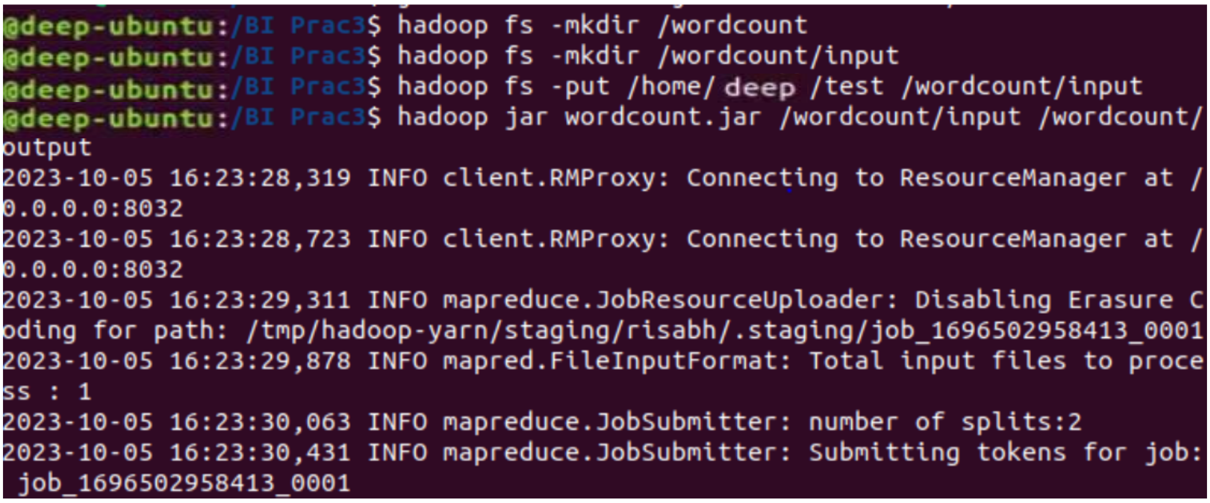
```
import java.io.IOException;
import java.util.Iterator;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
```

```
public class WordCountReducer extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {


    // Reduce method accepts the Key-Value pairs from mappers, does the aggregation based on keys

    // and produces the final output

    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {

        int sum = 0;


        // Iterates through all the values available with a key and adds them together, giving the final result as the key and the sum of its values

        while (values.hasNext()) {

            sum += values.next().get();

        }


        output.collect(key, new IntWritable(sum));

    }

}
```

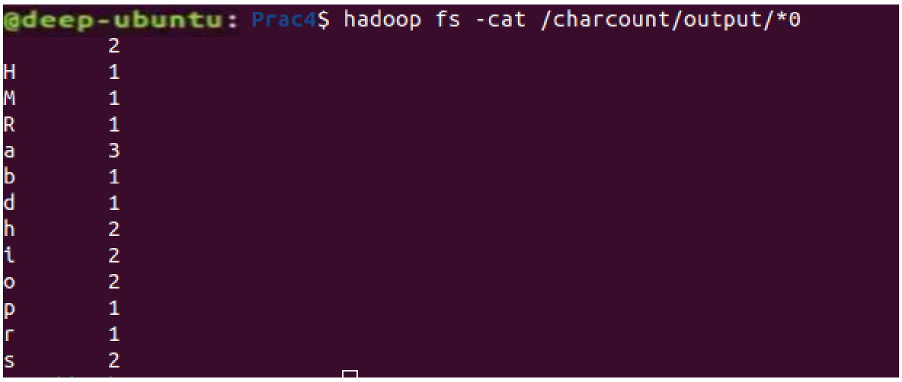
Output:



```
@deep-ubuntu:~$ cd 'BI Prac3'
@deep-ubuntu:~/BI Prac3$ export CLASSPATH="$HADOOP_HOME/share/hadoop/mapreduce/
hadoop-mapreduce-client-core-3.2.3.jar:$HADOOP_HOME/share/hadoop/mapreduce/hadoo
p-mapreduce-client-common-3.2.3.jar:$HADOOP_HOME/share/hadoop/common/hadoop-comm
on-3.2.3.jar"
@deep-ubuntu:~/BI Prac3$ javac -d . WordCount.java WordCountMapper.java WordCou
ntReducer.java
@deep-ubuntu:~/BI Prac3$ gedit Manifest.txt
@deep-ubuntu:~/BI Prac3$ jar cfm wordcount.jar Manifest.txt ./*.class
@deep-ubuntu:~/BI Prac3$
```



```
@deep-ubuntu:/BI Prac3$ hadoop fs -mkdir /wordcount
@deep-ubuntu:/BI Prac3$ hadoop fs -mkdir /wordcount/input
@deep-ubuntu:/BI Prac3$ hadoop fs -put /home/ deep /test /wordcount/input
@deep-ubuntu:/BI Prac3$ hadoop jar wordcount.jar /wordcount/input /wordcount/
output
2023-10-05 16:23:28,319 INFO client.RMProxy: Connecting to ResourceManager at /
0.0.0.0:8032
2023-10-05 16:23:28,723 INFO client.RMProxy: Connecting to ResourceManager at /
0.0.0.0:8032
2023-10-05 16:23:29,311 INFO mapreduce.JobResourceUploader: Disabling Erasure C
oding for path: /tmp/hadoop-yarn/staging/risabh/.staging/job_1696502958413_0001
2023-10-05 16:23:29,878 INFO mapred.FileInputFormat: Total input files to proce
ss : 1
2023-10-05 16:23:30,063 INFO mapreduce.JobSubmitter: number of splits:2
2023-10-05 16:23:30,431 INFO mapreduce.JobSubmitter: Submitting tokens for job:
 job_1696502958413_0001
```



```
@deep-ubuntu: Prac4$ hadoop fs -cat /charcount/output/*0
        2
H       1
M       1
R       1
a       3
b       1
d       1
h       2
i       2
o       2
p       1
r       1
s       2
```

Conclusion:

The provided Word Count program uses Hadoop MapReduce to efficiently count the occurrences of words in a text corpus. It is composed of a driver class, a mapper class, and a reducer class. The mapper tokenizes input text, emitting key-value pairs with words as keys and '1' as values. The reducer aggregates these counts and produces the final word count results. This program demonstrates the power of distributed processing for text analysis, with applications in tasks like text summarization and search engine indexing, efficiently handling large datasets.

Practical No 4

Aim: Write a map-reduce program to count the number of occurrences of each alphabetic character in the given dataset

Theory:

MapReduce is a programming model and data processing framework developed by Google and popularized by Apache Hadoop. It's designed for processing and generating large datasets that can be distributed across a cluster of computers. MapReduce simplifies the process of writing distributed and parallelized data processing applications. It consists of two main phases: the "Map" phase and the "Reduce" phase.

1. Map Phase:

- **Mapping:** In this phase, the input data is divided into chunks and processed in parallel. Each chunk is processed by a "mapper" function that extracts and emits key-value pairs based on the data's characteristics. The mapper function takes the input data, processes it, and emits intermediate key-value pairs.
- **Shuffling and Sorting:** After mapping, all emitted key-value pairs are shuffled and sorted by key. This process groups together all values associated with the same key. This prepares the data for the next phase.

2. Reduce Phase:

- **Reducing:** In this phase, a "reducer" function takes the sorted and grouped key-value pairs from the shuffle and sort phase. It then processes these key-value pairs to produce the final output. The reducer function can perform operations like aggregation, summarization, or any other computations based on the keys.

- **Output:** The final results are typically written to an output file or storage system.

Here are some key characteristics of MapReduce:

- Parallel Processing: MapReduce is designed to work in a distributed and parallelized manner. It processes data across multiple nodes in a cluster, which can significantly speed up data processing.

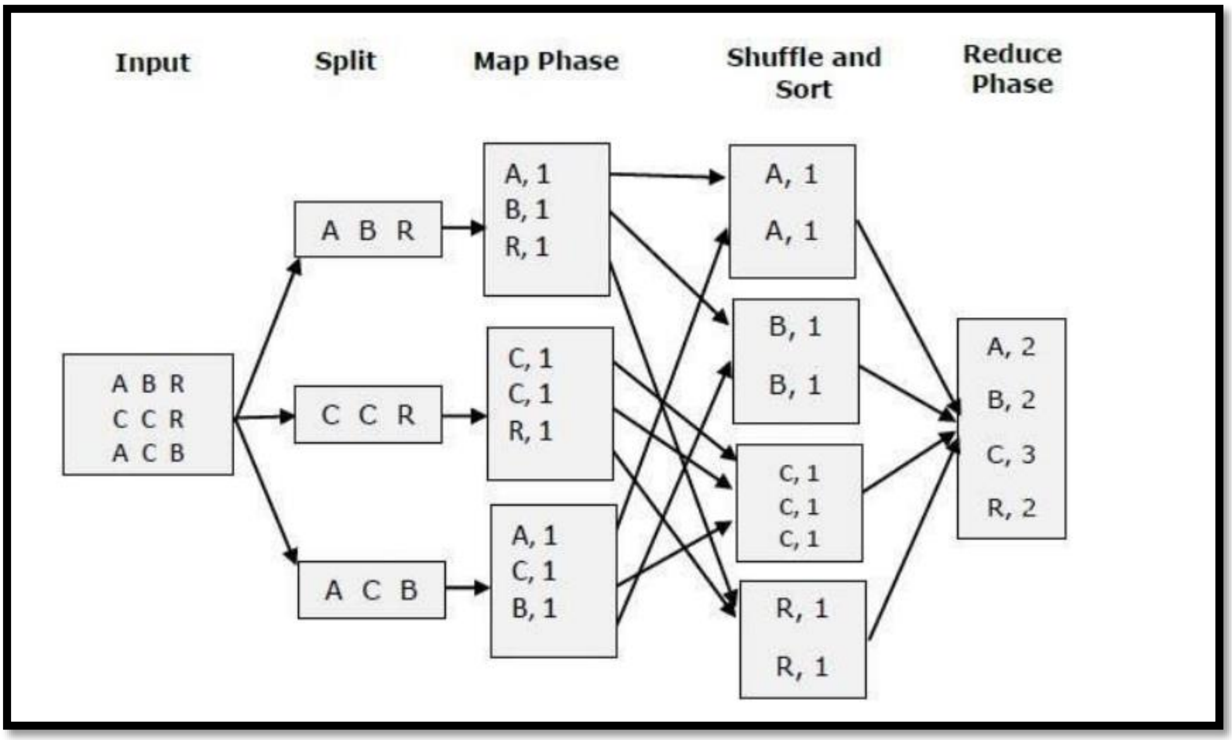
- Fault Tolerance: MapReduce frameworks like Hadoop provide fault tolerance. If a node in the cluster fails during processing, the framework can automatically reroute the work to other nodes, ensuring that the job continues to run.

- Scalability: MapReduce scales well with large datasets. You can add more nodes to your cluster to handle larger volumes of data.

- Simplicity: MapReduce abstracts many of the complexities of distributed processing, allowing developers to focus on writing the mapping and reducing functions without worrying about low-level distributed computing details.

MapReduce has been widely used for processing vast amounts of data, especially in big data and distributed computing environments. It's a foundational concept in many big data ecosystems, including Hadoop, and has been the basis for a wide range of data processing tasks, including log analysis, data transformation, and machine learning.

Character Count Example:



Code:

WC_Runner.java(Driver Class)

```
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.TextOutputFormat;

public class WC_Runner {
    public static void main(String[] args) throws IOException {
        JobConf conf = new JobConf(WC_Runner.class);
        conf.setJobName("CharCount");
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);
        conf.setMapperClass(WC_Mapper.class);
        conf.setCombinerClass(WC_Reducer.class);
        conf.setReducerClass(WC_Reducer.class);
        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);
        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));
        JobClient.runJob(conf);
    }
}
```

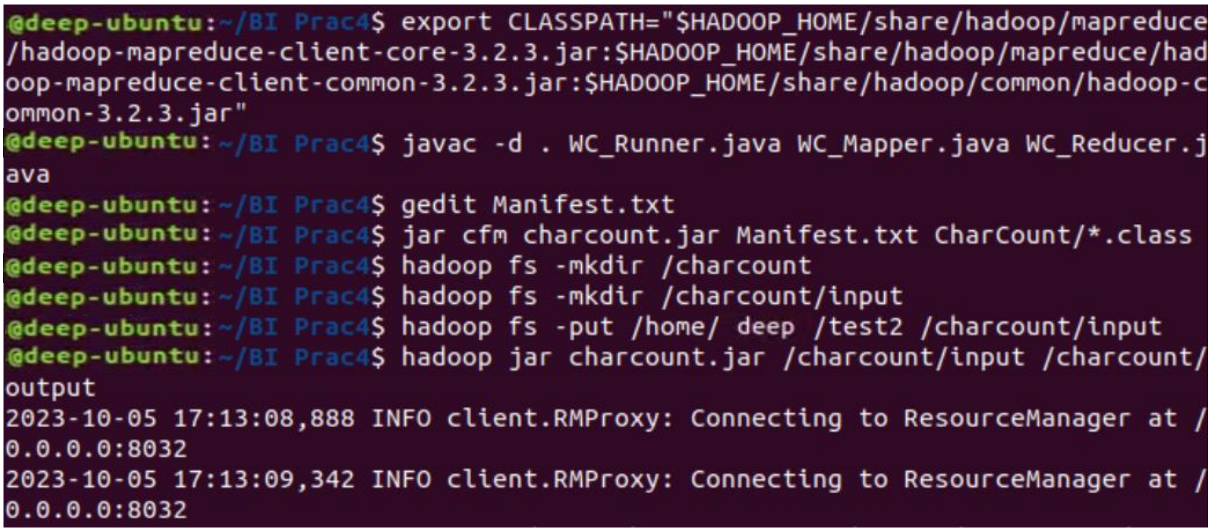
WC_Mapper.java(Mapper Class)

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

public class WC_Mapper extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
        String line = value.toString();
        String[] tokenizer = line.split("");
        for (String singleChar : tokenizer) {
            Text charKey = new Text(singleChar);
            IntWritable one = new IntWritable(1);
            output.collect(charKey, one);
        }
    }
}
```

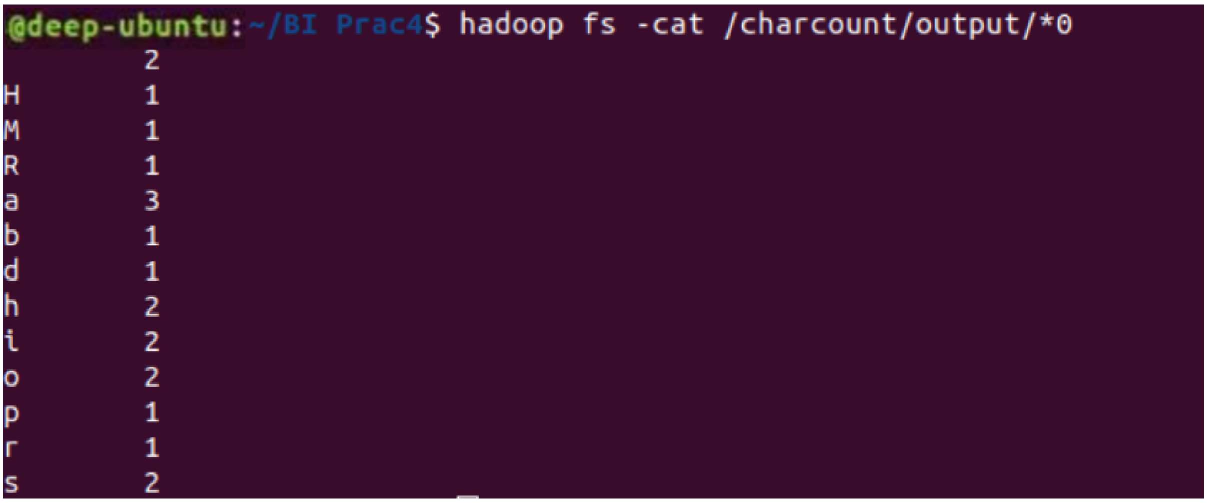
WC_Reducer.java(Reducer Class)

```
import java.io.IOException;
import java.util.Iterator;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;

public class WC_Reducer extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```

Output:



```
@deep-ubuntu:~/BI Prac4$ export CLASSPATH="$HADOOP_HOME/share/hadoop/mapreduce
/hadoop-mapreduce-client-core-3.2.3.jar:$HADOOP_HOME/share/hadoop/mapreduce/had
oop-mapreduce-client-common-3.2.3.jar:$HADOOP_HOME/share/hadoop/common/hadoop-c
ommon-3.2.3.jar"
@deep-ubuntu:~/BI Prac4$ javac -d . WC_Runner.java WC_Mapper.java WC_Reducer.j
ava
@deep-ubuntu:~/BI Prac4$ gedit Manifest.txt
@deep-ubuntu:~/BI Prac4$ jar cfm charcount.jar Manifest.txt CharCount/*.class
@deep-ubuntu:~/BI Prac4$ hadoop fs -mkdir /charcount
@deep-ubuntu:~/BI Prac4$ hadoop fs -mkdir /charcount/input
@deep-ubuntu:~/BI Prac4$ hadoop fs -put /home/ deep /test2 /charcount/input
@deep-ubuntu:~/BI Prac4$ hadoop jar charcount.jar /charcount/input /charcount/
output
2023-10-05 17:13:08,888 INFO client.RMProxy: Connecting to ResourceManager at /
0.0.0.0:8032
2023-10-05 17:13:09,342 INFO client.RMProxy: Connecting to ResourceManager at /
0.0.0.0:8032
```



```
@deep-ubuntu:~/BI Prac4$ hadoop fs -cat /charcount/output/*0
        2
H       1
M       1
R       1
a       3
b       1
d       1
h       2
i       2
o       2
p       1
r       1
s       2
```

Conclusion:

The given Java program, consisting of a driver class , a mapper class , and a reducer class , implements a simple MapReduce job for counting the occurrences of characters in input text data.

The Charcount program is a Hadoop MapReduce job that processes text data. It begins by configuring the job and setting the input and output file formats. The Charmap class, acting as the mapper, tokenizes the input text, emits key-value pairs for each character found in the text, and counts them. The Charreduce class, serving as the reducer, sums the character counts and emits the result .

This MapReduce job provides a simple example of parallelizing character counting in large text datasets. It demonstrates the use of Hadoop's MapReduce framework for distributed data processing.

Practical No 5

Aim: Write a map-reduce program to determine the average ratings of movies. The input consists of a series of lines, each containing a movie number, user number, rating and a timestamp.

Theory:

MapReduce is a programming model and data processing framework developed by Google and popularized by Apache Hadoop. It's designed for processing and generating large datasets that can be distributed across a cluster of computers. MapReduce simplifies the process of writing distributed and parallelized data processing applications. It consists of two main phases: the "Map" phase and the "Reduce" phase.

1. Map Phase:

- **Mapping:** In this phase, the input data is divided into chunks and processed in parallel. Each chunk is processed by a "mapper" function that extracts and emits key-value pairs based on the data's characteristics. The mapper function takes the input data, processes it, and emits intermediate key-value pairs.
- **Shuffling and Sorting:** After mapping, all emitted key-value pairs are shuffled and sorted by key. This process groups together all values associated with the same key. This prepares the data for the next phase.

2. Reduce Phase:

- **Reducing:** In this phase, a "reducer" function takes the sorted and grouped key-value pairs from the shuffle and sort phase. It then processes these key-value pairs to produce the final output.

The reducer function can perform operations like aggregation, summarization, or any other computations based on the keys.

- **Output:** The final results are typically written to an output file or storage system.

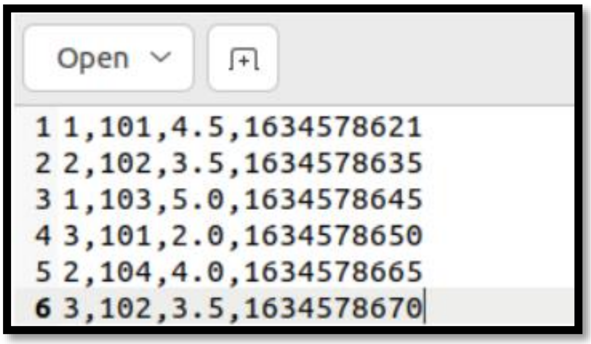
Code:

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.*;

import org.apache.hadoop.mapreduce.*;


public class MovieRatingsAverage {

   public static class TokenizerMapper extends Mapper<Object, Text, Text, DoubleWritable>
{

      private Text movieId = new Text();

      private DoubleWritable rating = new DoubleWritable();


      public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {

         String[] fields = value.toString().split(",");

         if (fields.length == 4) {

            movieId.set(fields[0]);

            rating.set(Double.parseDouble(fields[2]));

            context.write(movieId, rating);

         }

      }

   }
```

```
  public static class DoubleSumReducer extends Reducer<Text, DoubleWritable, Text,
DoubleWritable> {
    private DoubleWritable result = new DoubleWritable();

    public void reduce(Text key, Iterable<DoubleWritable> values, Context context)
        throws IOException, InterruptedException {
      double sum = 0;
      int count = 0;
      for (DoubleWritable val : values) {
        sum += val.get();
        count++;
      }
      double average = sum / count;
      result.set(average);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "Movie Ratings Average");
    job.setJarByClass(MovieRatingsAverage.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(DoubleSumReducer.class);
    job.setReducerClass(DoubleSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(DoubleWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
```

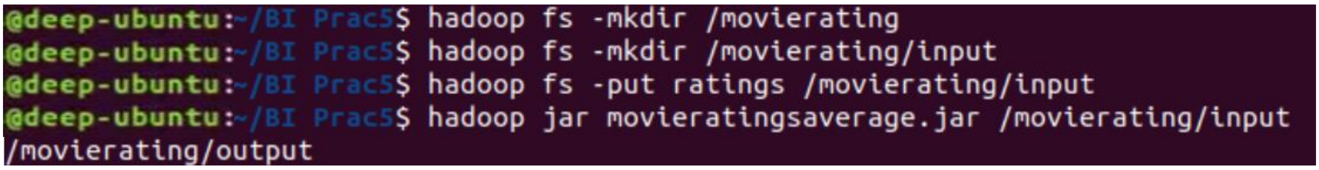
}

Ratings.txt

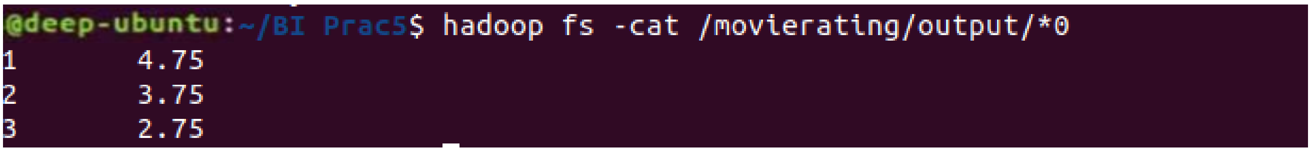


```
1,101,4.5,1634578621
2,102,3.5,1634578635
1,103,5.0,1634578645
3,101,2.0,1634578650
2,104,4.0,1634578665
3,102,3.5,1634578670
```

Output:



```
@deep-ubuntu:~/BI Prac5$ hadoop fs -mkdir /movierating
@deep-ubuntu:~/BI Prac5$ hadoop fs -mkdir /movierating/input
@deep-ubuntu:~/BI Prac5$ hadoop fs -put ratings /movierating/input
@deep-ubuntu:~/BI Prac5$ hadoop jar movieratingsaverage.jar /movierating/input
/movierating/output
```



```
@deep-ubuntu:~/BI Prac5$ hadoop fs -cat /movierating/output/*0
1       4.75
2       3.75
3       2.75
```

Conclusion:

The provided Java Map-Reduce program efficiently calculates the average ratings of movies from a dataset. It consists of a Mapper class that extracts movie ratings, a Reducer class that computes the average rating for each movie, and a Runner class to configure and execute the Hadoop job. This program demonstrates the power of distributed data processing, allowing it to handle large datasets. By running this code on a Hadoop cluster, it enables the analysis of movie ratings at scale and can provide valuable insights for recommendation systems or data analysis in the context of movie ratings.

Practical No 6

Aim: Write a map-reduce program: (i) to find matrix-vector multiplication; (ii) to compute selections and projections; (iii) to find union, intersection, difference, natural Join for a given dataset.

Theory:

MapReduce is a programming model and data processing framework developed by Google and popularized by Apache Hadoop. It's designed for processing and generating large datasets that can be distributed across a cluster of computers. MapReduce simplifies the process of writing distributed and parallelized data processing applications. It consists of two main phases: the "Map" phase and the "Reduce" phase.

1. Map Phase:

- **Mapping:** In this phase, the input data is divided into chunks and processed in parallel. Each chunk is processed by a "mapper" function that extracts and emits key-value pairs based on the data's characteristics. The mapper function takes the input data, processes it, and emits intermediate key-value pairs.
- **Shuffling and Sorting:** After mapping, all emitted key-value pairs are shuffled and sorted by key. This process groups together all values associated with the same key. This prepares the data for the next phase.

2. Reduce Phase:

- **Reducing:** In this phase, a "reducer" function takes the sorted and grouped key-value pairs from the shuffle and sort phase. It then processes these key-value pairs to produce the final output.

The reducer function can perform operations like aggregation, summarization, or any other computations based on the keys.

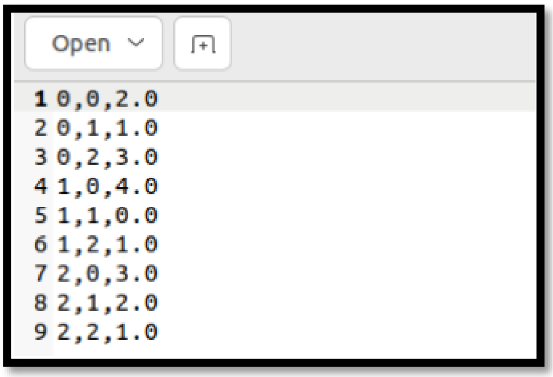
- **Output:** The final results are typically written to an output file or storage system.

Code:

(i) To find matrix-vector multiplication

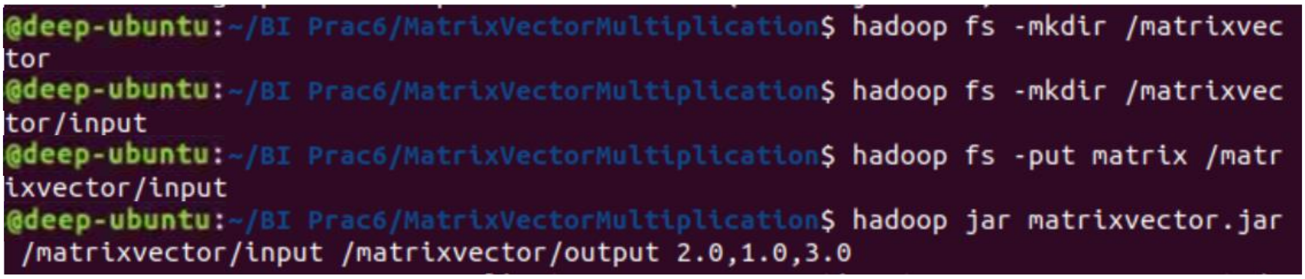
```
import java.io.IOException;

import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.*;

import org.apache.hadoop.mapreduce.*;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;


public class MatrixVectorMultiplication {

    public static class TokenizerMapper extends Mapper<Object, Text, IntWritable,
DoubleWritable> {

        private IntWritable rowIndex = new IntWritable();

        private DoubleWritable result = new DoubleWritable();

        private double[] vector;


        public void setup(Context context) throws IOException, InterruptedException {

            Configuration conf = context.getConfiguration();

            String vectorStr = conf.get("vector");

            String[] vectorTokens = vectorStr.split(",");

            vector = new double[vectorTokens.length];

            for (int i = 0; i < vectorTokens.length; i++) {
                vector[i] = Double.parseDouble(vectorTokens[i]);
            }
        }

        public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
            StringTokenizer tokenizer = new StringTokenizer(value.toString(), ",");
            int row = Integer.parseInt(tokenizer.nextToken());
            int col = Integer.parseInt(tokenizer.nextToken());
            double matrixValue = Double.parseDouble(tokenizer.nextToken());

            // Perform the multiplication and emit the result
            double product = matrixValue * vector[col];
            rowIndex.set(row);
            result.set(product);
            context.write(rowIndex, result);
        }
    }

    public static class DoubleSumReducer extends Reducer<IntWritable, DoubleWritable,
IntWritable, DoubleWritable> {
        private DoubleWritable result = new DoubleWritable();

        public void reduce(IntWritable key, Iterable<DoubleWritable> values, Context
context)
                throws IOException, InterruptedException {
            double sum = 0;
            for (DoubleWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        conf.set("vector", args[2]); // Pass the vector as a comma-separated string

        Job job = Job.getInstance(conf, "Matrix-Vector Multiplication");
        job.setJarByClass(MatrixVectorMultiplication.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(DoubleSumReducer.class);
        job.setReducerClass(DoubleSumReducer.class);
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(DoubleWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0])); // Matrix input
        FileOutputFormat.setOutputPath(job, new Path(args[1])); // Output
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Matrix.txt

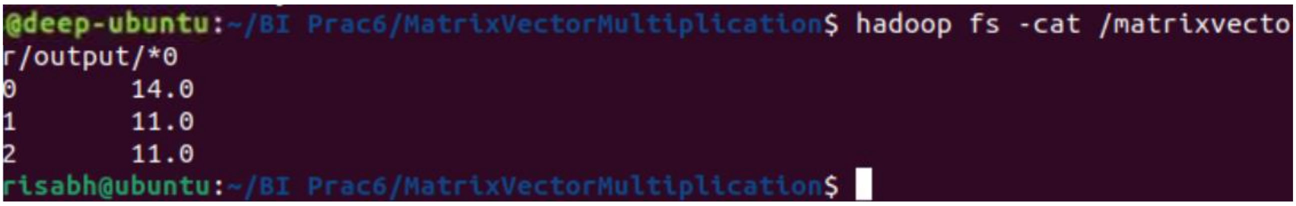


```
0,0,2.0
0,1,1.0
0,2,3.0
1,0,4.0
1,1,0.0
1,2,1.0
2,0,3.0
2,1,2.0
2,2,1.0
```

Output:



```
@deep-ubuntu:~/BI Prac6/MatrixVectorMultiplication$ hadoop fs -mkdir /matrixvec
tor
@deep-ubuntu:~/BI Prac6/MatrixVectorMultiplication$ hadoop fs -mkdir /matrixvec
tor/input
@deep-ubuntu:~/BI Prac6/MatrixVectorMultiplication$ hadoop fs -put matrix /matr
ixvector/input
@deep-ubuntu:~/BI Prac6/MatrixVectorMultiplication$ hadoop jar matrixvector.jar
 /matrixvector/input /matrixvector/output 2.0,1.0,3.0
```



```
@deep-ubuntu:~/BI Prac6/MatrixVectorMultiplication$ hadoop fs -cat /matrixvecto
r/output/*0
0       14.0
1       11.0
2       11.0
risabh@ubuntu:~/BI Prac6/MatrixVectorMultiplication$
```

Conclusion:

The Mapper class retrieves the vector as a configuration parameter and multiplies matrix elements with the corresponding vector values, emitting key-value pairs. The Reducer class then sums up the products to calculate the final result. The main method configures the job, specifying input and output paths, and runs the MapReduce job.

In conclusion, this code efficiently performs matrix-vector multiplication using Hadoop's MapReduce framework, demonstrating the power of parallel computing for large-scale data processing.

Practical No 7

Aim: Write a program to construct different types of k-shingles for given document.

Theory:

K-shingling is a technique used in text analysis and document similarity measurement. It involves breaking a text document or a sequence of text into smaller, fixed-length chunks or "shingles" of length K. These shingles are typically created by sliding a fixed-size window of K words or characters through the text, extracting the content within the window at each position.

Here's an example to illustrate K-shingling with a simple sentence:

Original Sentence: "The quick brown fox jumps over the lazy dog."

Let's say we want to create 3-shingles (K = 3) from this sentence. We'll slide a 3-word window through the sentence to extract the shingles:

1. "The quick brown"
2. "quick brown fox"
3. "brown fox jumps"
4. "fox jumps over"
5. "jumps over the"
6. "over the lazy"
7. "the lazy dog."

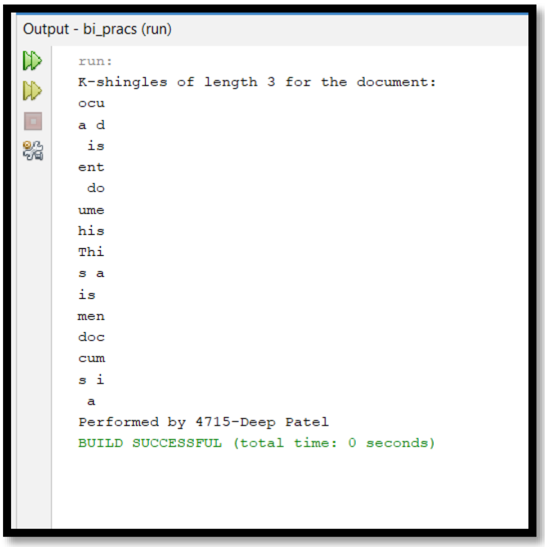
These are the 3-shingles extracted from the original sentence. You can see that each shingle is a consecutive set of 3 words from the sentence. K-shingling can be useful in various natural language processing tasks, including text similarity measurement, plagiarism detection, and document clustering, as it helps represent the text data in a structured way for analysis.

Code:

```
package bi_pracs;
import java.util.HashSet;
import java.util.Set;

public class KShingles {
    public static void main(String[] args) {
        String document = "This is a document";
        int k = 3; // Change k to the desired value

        Set<String> shingles = constructKShingles(document, k);

        System.out.println("K-shingles of length " + k + " for the document:");
        for (String shingle : shingles) {
            System.out.println(shingle);
        }
        System.out.println("Performed by 4715-Deep Patel");
    }

    public static Set<String> constructKShingles(String document, int k) {
        Set<String> shingles = new HashSet<>();

        // Iterate through the document and create k-shingles
        for (int i = 0; i <= document.length() - k; i++) {
            String shingle = document.substring(i, i + k);
            shingles.add(shingle);
        }
        return shingles;
    }
}
```

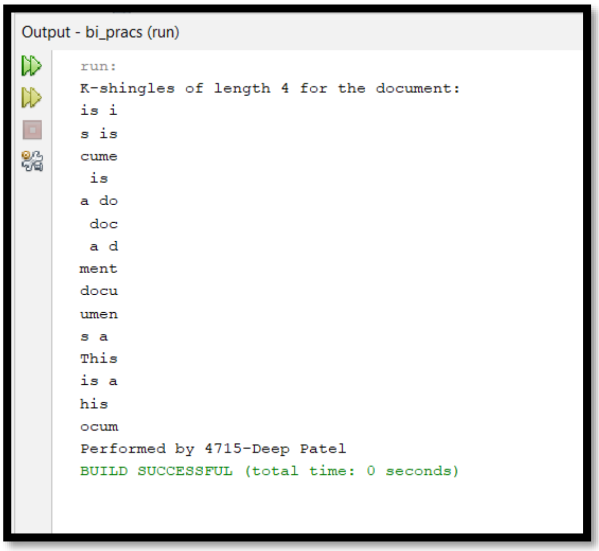
Output:

For k=3



```
run:
K-shingles of length 3 for the document:
ocu
a d
 is
ent
 do
ume
his
Thi
s a
is 
men
doc
cum
s i
 a 
Performed by 4715-Deep Patel
BUILD SUCCESSFUL (total time: 0 seconds)
```

For k=4



```
run:
K-shingles of length 4 for the document:
is i
s is
cume
 is 
a do
 doc
 a d
ment
docu
umen
s a 
This
is a
his 
ocum
Performed by 4715-Deep Patel
BUILD SUCCESSFUL (total time: 0 seconds)
```

Conclusion:

Overall, this program demonstrates how to generate K-shingles from a given text document, making it useful for various text analysis tasks such as plagiarism detection, document similarity measurement, and more. You can customize the input document and 'k' value to adapt it to your specific use case.

Practical No 8

Aim: Write a program for measuring similarity among documents and detecting passages which have been reused.

Theory:

Jaccard similarity is a measure used to calculate the similarity between two sets by comparing their intersection and union. It is particularly useful in situations where you want to determine how similar two sets are based on the elements they contain.

The Jaccard similarity (J) is defined as the size of the intersection of two sets divided by the size of their union:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Here's an example to illustrate Jaccard similarity:

Suppose you have two sets, Set A and Set B, representing the items in two documents:

Set A = {apple, banana, orange, grape}

Set B = {banana, orange, watermelon, kiwi}

To calculate the Jaccard similarity between Set A and Set B:

1. Find the intersection of the sets (i.e., the common elements):

A∩B = {banana, orange}

2. Find the union of the sets (i.e., all unique elements from both sets):

AUB = {apple, banana, orange, grape, watermelon, kiwi}

3. Calculate the Jaccard similarity using the formula:

J(A, B) =|A∩B|/|AUB|=2/6=0.33

So, the Jaccard similarity between Set A and Set B is 0.33 (or 33%).

In this example, the Jaccard similarity measures how similar the sets are based on the presence of common elements (banana and orange) relative to the total number of unique elements in both sets. A higher Jaccard similarity indicates greater similarity between the sets, while a lower Jaccard similarity suggests less similarity.

Code:

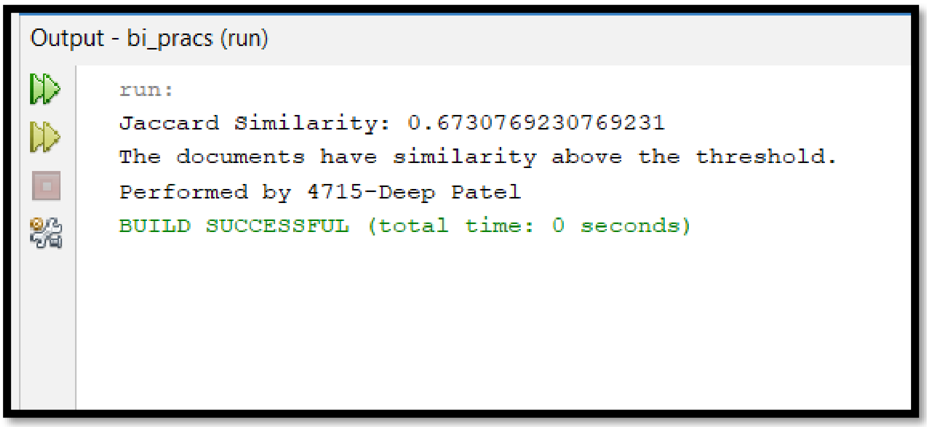
```
package bi_pracs;
import java.util.HashSet;
import java.util.Set;

public class DocumentSimilarity {
   public static void main(String[] args) {
      String document1 = "This is document one. It contains some text.";
      String document2 = "This is document two. It also contains some text.";

      int k = 3; // Change k to the desired value for k-shingles
```

```
        Set<String> shingles1 = constructKShingles(document1, k);
        Set<String> shingles2 = constructKShingles(document2, k);

        double jaccardSimilarity = calculateJaccardSimilarity(shingles1, shingles2);
        System.out.println("Jaccard Similarity: " + jaccardSimilarity);

        if (jaccardSimilarity > 0.5) {
            System.out.println("The documents have similarity above the threshold.");
            // You can add code here to identify the reused passages.
        } else {
            System.out.println("The documents have low similarity.");
        }
        System.out.println("Performed by 4715-Deep Patel");
    }

    public static Set<String> constructKShingles(String document, int k) {
        Set<String> shingles = new HashSet<>();

        for (int i = 0; i <= document.length() - k; i++) {
            String shingle = document.substring(i, i + k);
            shingles.add(shingle);
        }
        return shingles;
    }

    public static double calculateJaccardSimilarity(Set<String> set1, Set<String> set2) {
        int intersectionSize = 0;
        int unionSize = set1.size() + set2.size();

        for (String shingle : set1) {
            if (set2.contains(shingle)) {
                intersectionSize++;
                unionSize--;
            }
        }
        return (double) intersectionSize / unionSize;
    }
}
```

Output:

Document1 = "This is document one. It contains some text.";

Document2 = "This is document two. It also contains some text.";



Output - bi_pracs (run)

```
run:
Jaccard Similarity: 0.6730769230769231
The documents have similarity above the threshold.
Performed by 4715-Deep Patel
BUILD SUCCESSFUL (total time: 0 seconds)
```

Conclusion:

The provided Java program calculates the Jaccard similarity between two text documents using K-shingles. It allows users to specify the shingle length 'k' and checks whether the similarity between the documents exceeds a predefined threshold (0.5 by default). This program is useful for identifying similarities between documents, making it applicable in tasks like plagiarism detection and document clustering. It offers flexibility for customization and provides a clear indication of document similarity, making it a valuable tool for text analysis.

Practical No 9

Aim: Write a program to compute the n-moment for a given stream where n is given.

Theory:

In statistics, moments which are statistical measures traditionally used for analyzing continuous numerical data distributions, to a categorical data stream. Here's a more concise explanation of this concept:

1. Data Stream: The program processes a data stream consisting of categorical values (e.g., "a," "b," "c") rather than continuous numerical values.

2. Moments: In statistics, moments are usually used to describe characteristics of continuous probability distributions, such as mean, variance, skewness, and kurtosis.

3. Unconventional Application: This program adapts the concept of moments to the categorical data stream by calculating:

- Zeroth Moment: This counts the number of unique categories in the stream, essentially measuring the diversity or distinct elements within the data.

- First Moment: It sums the lengths of segments of identical consecutive elements, representing the total length of the stream.

- Second Moment: This sums the squares of the lengths of segments of identical consecutive elements, which can provide insights into the distribution's spread.

4. ArrayList: The program uses an ArrayList to store the lengths of segments with identical consecutive elements.

5. Sorting: The data stream is sorted to group identical elements together, making it easier to calculate the moments.

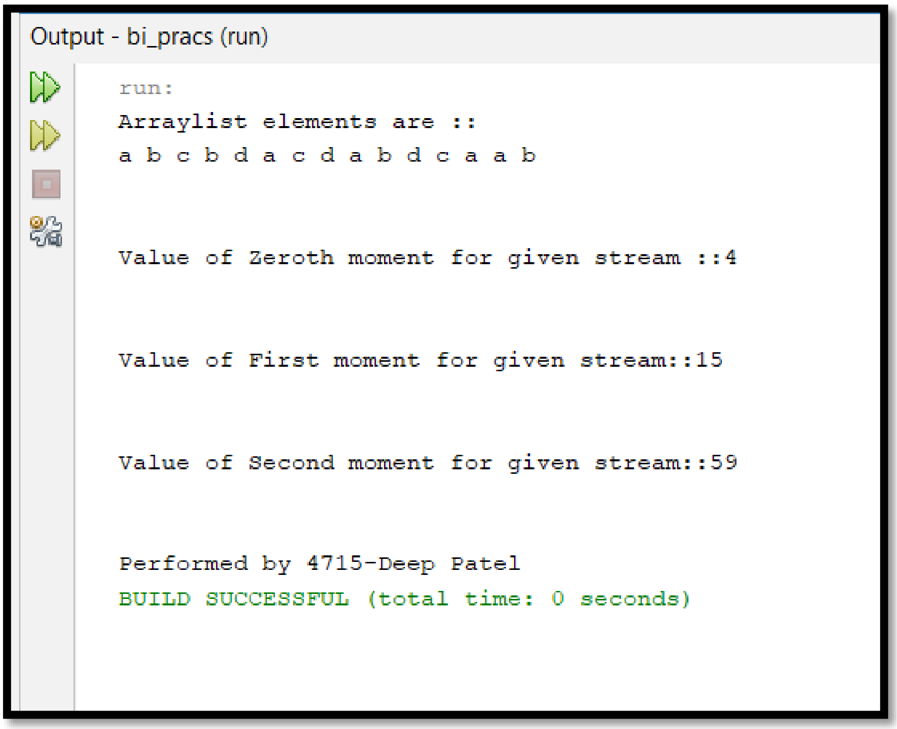
Code:

```
package bi_pracs;
import java.io.*;
import java.util.*;

class n_moment2
{
   public static void main(String args[])
   {
      int n = 15;
      String stream[] = {"a", "b", "c", "b", "d", "a", "c", "d", "a", "b", "d", "c", "a", "a", "b"};
      int zero_moment = 0, first_moment = 0, second_moment = 0, count = 1;
      ArrayList<Integer> arrlist = new ArrayList<>();
      System.out.println("Arraylist elements are :: ");
      for (int i = 0; i < 15; i++)
      {
         System.out.print(stream[i] + " ");
      }
```

```
Arrays.sort(stream);
// Calculate Zeroth moment (calculates unique elements - raised to zero)
for (int i = 1; i < n; i++)
{
   if (stream[i].equals(stream[i - 1]))
   {
      count++;
   }
   else
   {
      arrlist.add(count);
      count = 1;
   }
}
arrlist.add(count);
zero_moment = arrlist.size();
System.out.println("\n\n\nValue of Zeroth moment for given stream ::" + zero_moment);
// Calculate First moment (Calculate length of the stream - raised to one)
for (int i = 0; i < arrlist.size(); i++)
{
   first_moment += arrlist.get(i);
}
System.out.println("\n\nValue of First moment for given stream::" + first_moment);
// Calculate Second moment (raised to two)
for (int i = 0; i < arrlist.size(); i++)
{
   int j = arrlist.get(i);
   second_moment += (j * j);
}
System.out.println("\n\nValue of Second moment for given stream::" + second_moment);
```

```
        System.out.println("\n\nPerformed by 4715-Deep Patel");
    }
}
```

Output:



```
run:
Arraylist elements are ::
a b c b d a c d a b d c a a b


Value of Zeroth moment for given stream ::4


Value of First moment for given stream::15


Value of Second moment for given stream::59


Performed by 4715-Deep Patel
BUILD SUCCESSFUL (total time: 0 seconds)
```

Conclusion:

In summary, this concept is to apply the idea of moments, which is traditionally used for continuous data, to a categorical data stream. It measures different characteristics of the categorical data, such as diversity (zeroth moment), total length (first moment), and spread (second moment), using a somewhat unconventional approach. While this approach may not have direct statistical significance in the traditional sense, it showcases how statistical concepts can be adapted and applied in various contexts.

Practical No 10

Aim: Write a program to demonstrate the Alon-Matias-Szegedy Algorithm for second moments.

Theory:

The Alon-Matias-Szegedy (AMS) Algorithm is a randomized algorithm used for approximating the second moment of a data stream. The second moment is often referred to as the "variance" in statistics and provides a measure of the spread or dispersion of data.

The AMS algorithm is particularly useful when dealing with large data streams or when memory resources are limited, as it allows you to estimate the second moment without storing the entire data stream in memory. It provides a probabilistic estimate of the second moment with a controlled error rate.

Here's a brief overview of how the AMS Algorithm works:

1)Initialization: Initialize a set of random variables (hash functions) that map elements from the data stream to small values (usually 1 or -1).

2)Processing Data Stream: As you process the data stream element by element, apply the hash functions to each element and accumulate the results.

3)Estimating the Second Moment: After processing the entire data stream, square the accumulated sum and take the average of these squared values. This average is an estimate of the second moment of the data stream.

The key insight behind the AMS algorithm is that the squared sum of random variables provides an unbiased estimate of the second moment. By using multiple hash functions, the algorithm reduces the variance of the estimate, resulting in a more accurate approximation of the second moment.

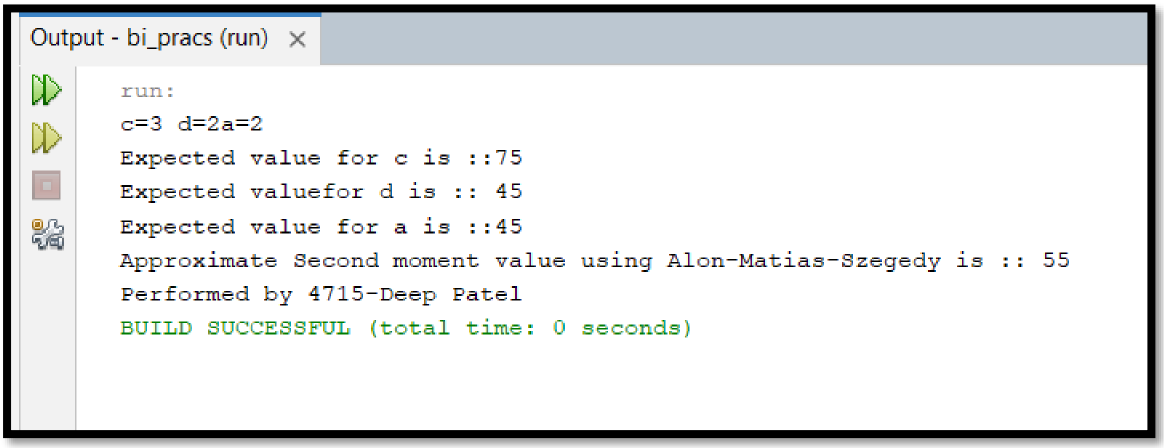
One of the advantages of the AMS algorithm is that it can provide reasonably accurate estimates with a relatively small number of hash functions, making it memory-efficient and suitable for streaming data scenarios.

AMS algorithm is commonly used in data streaming applications, network monitoring, and other situations where real-time estimation of variance or second moments is required, and memory is limited. However, it's important to note that the accuracy of the estimate depends on the number of hash functions used and the chosen error rate, so careful parameter selection is necessary for specific use cases.

Code:

```
package bi_pracs;
import java.io.*;
import java.util.*;

class AMSA
{
public static int findCharCount(String stream,char XE,int random,int n) {
int countOccurance=0;
for(int i=random;i<n;i++)
{
if(stream.charAt(i)==XE)
{
countOccurance++;
//System.out.println(countOccurance+" "+i);
}
}
return countOccurance;
}
public static int estimateValue(int XV1,int n)
{
int ExpValue;
ExpValue=n*(2*XV1-1);
return ExpValue;
}
public static void main(String args[])
{
int n=15;
String stream="abcbdacdabdcaab";
int random1=3,random2=8,random3=13;
char XE1,XE2,XE3;
int XV1,XV2,XV3;
int ExpValuXE1, ExpValuXE2, ExpValuXE3;
int apprSecondMomentValue;
XE1=stream.charAt(random1-1);
XE2=stream.charAt(random2-1);
XE3=stream.charAt(random3-1);
//System.out.println(XE1+" "+XE2+" "+XE3);
XV1=findCharCount(stream,XE1,random1-1,n);
XV2=findCharCount(stream,XE2,random2-1,n);
XV3=findCharCount(stream,XE3,random3-1,n);
System.out.println(XE1+"="+XV1+" "+XE2+"="+XV2+""+XE3+"="+XV3);
ExpValuXE1=estimateValue(XV1,n);
ExpValuXE2=estimateValue(XV2,n);
ExpValuXE3=estimateValue(XV3,n);
System.out.println("Expected value for "+XE1+" is ::"+ExpValuXE1);
System.out.println("Expected valuefor "+XE2+" is :: "+ExpValuXE2);
System.out.println("Expected value for "+XE3+" is ::"+ExpValuXE3);
apprSecondMomentValue=(ExpValuXE1+ExpValuXE2+ExpValuXE3)/3;
System.out.println("Approximate Second moment value using Alon-Matias-Szegedy is :: "+apprSecondMomentValue);
System.out.println("Performed by 4715-Deep Patel");
}
}
```

Output:



```
run:
c=3 d=2a=2
Expected value for c is ::75
Expected valuefor d is :: 45
Expected value for a is ::45
Approximate Second moment value using Alon-Matias-Szegedy is :: 55
Performed by 4715-Deep Patel
BUILD SUCCESSFUL (total time: 0 seconds)
```

Conclusion:

The provided Java program implements the Alon-Matias-Szegedy (AMS) algorithm to estimate the second moment of a given data stream. It selects three random positions within the stream and counts the occurrences of specific characters. Then, it calculates the expected values for these characters and averages them to approximate the second moment. The program offers a practical example of how the AMS algorithm can be applied to analyze data distribution characteristics, making it useful for scenarios with large data streams and limited memory resources.