EXPERIMENT-01

DATA VISUALIZATION USING DIFFERENT FORMS

AIM:- Data Visualization using Scatter Plot, Pie Chart, Bars Plot and Histograms.

THEORY:-

Scatter Plot:

Theory: A scatter plot is used to visualize the relationship between two continuous variables. Each point represents an observation in the dataset, with the x-axis showing one variable and the y-axis showing the other. Scatter plots are useful for identifying patterns, trends, and correlations between variables.

Example: Visualizing the relationship between the number of hours studied and exam scores for a group of students.

Pie Chart:

Theory: A pie chart is a circular statistical graphic divided into slices to illustrate numerical proportions. Each slice represents a proportion of the whole, and the size of each slice is proportional to the quantity it represents. Pie charts are useful for showing the composition of a categorical variable.

Example: Showing the distribution of different types of crimes in a city, where each slice represents the percentage of each type of crime.

Bar Plot:

Theory: A bar plot (or bar chart) is a graphical display of data using rectangular bars with lengths proportional to the values they represent. Bar plots are used to compare different categories or groups. The height (or length) of each bar represents the value of the variable.

Example: Comparing the sales performance of different products in a store by plotting the total sales of each product as bars.

Histogram:

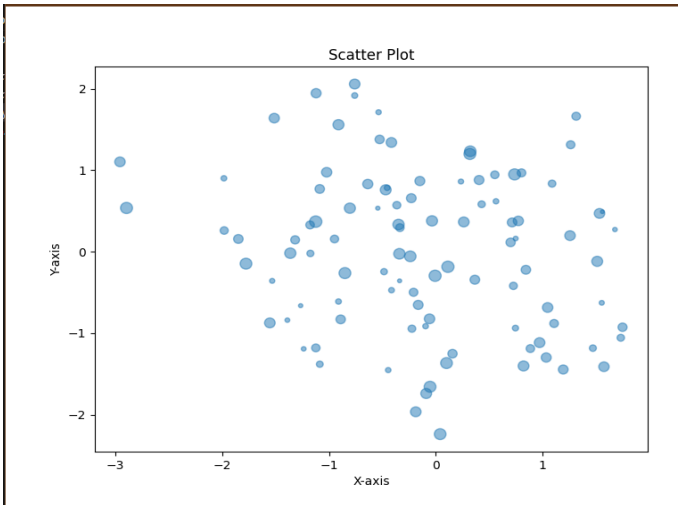
Theory: A histogram is a graphical representation of the distribution of numerical data. It consists of a series of adjacent rectangles (bins) with bases on the x-axis and heights proportional to the frequency of occurrences in each bin. Histograms are useful for understanding the underlying distribution of a continuous variable.

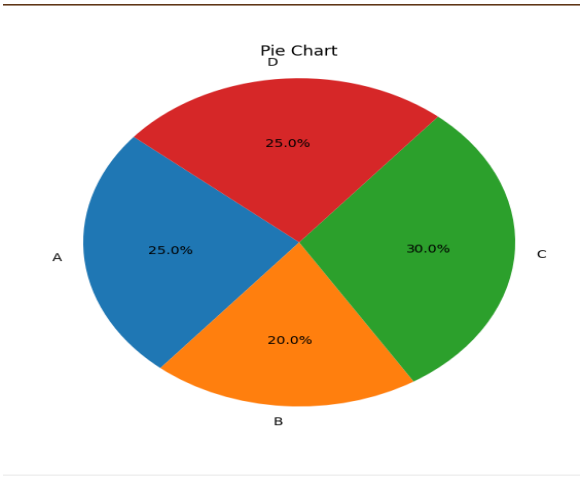
Example: Analyzing the distribution of ages in a population by dividing the ages into bins (e.g., 0-10, 11-20, 21-30, etc.) and plotting the frequency of individuals in each age group.

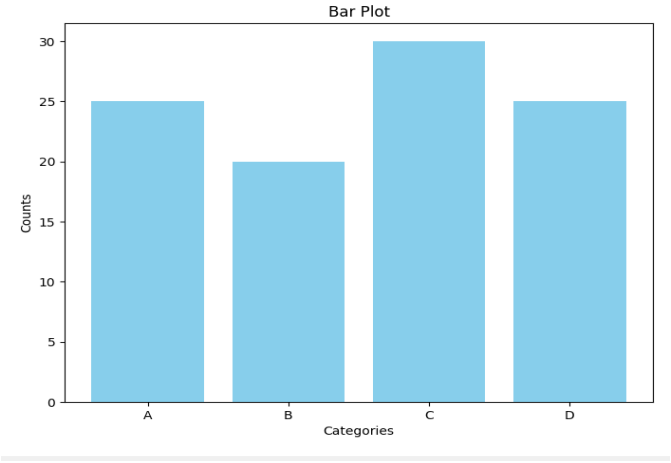
CODE:-

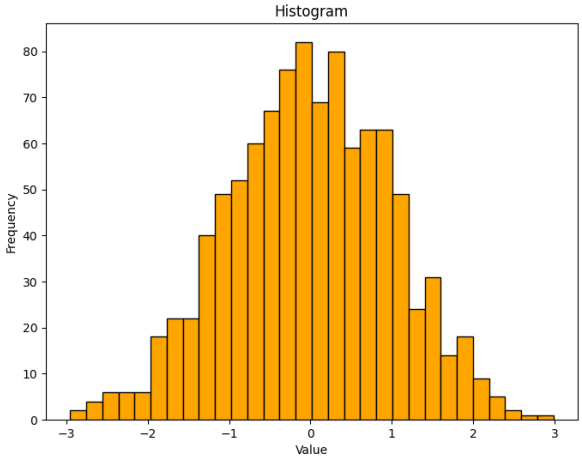
```
import matplotlib.pyplot as plt
import numpy as np

# Sample data
x = np.random.randn(100)
y = np.random.randn(100)
sizes = np.random.randint(10, 100, size=100)
categories = ['A', 'B', 'C', 'D']
counts = [25, 20, 30, 25]
hist_data = np.random.randn(1000)

# Scatter plot
plt.figure(figsize=(8, 6))
plt.scatter(x, y, s=sizes, alpha=0.5)
plt.title('Scatter Plot')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.show()

# Pie chart
plt.figure(figsize=(8, 6))
plt.pie(counts, labels=categories, autopct='%1.1f%%', startangle=140)
plt.title('Pie Chart')
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.show()

# Bar plot
plt.figure(figsize=(8, 6))
plt.bar(categories, counts, color='skyblue')
plt.title('Bar Plot')
plt.xlabel('Categories')
plt.ylabel('Counts')
plt.show()

# Histogram
plt.figure(figsize=(8, 6))
plt.hist(hist_data, bins=30, color='orange', edgecolor='black')
plt.title('Histogram')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.show()
```

OUTPUT:-


Scatter Plot
Y-axis
X-axis
2
1
0
−1
−2
−3
−2
−1
0
1


Pie Chart
D
25.0%
A
25.0%
30.0%
C
20.0%
B


Bar Plot
Counts
Categories
30
25
20
15
10
5
0
A
B
C
D


Histogram
Frequency
Value
80
70
60
50
40
30
20
10
0
−3
−2
−1
0
1
2
3

EXPERIMENT-02

MEASURE OF CENTRAL TENDENCY AND MEASURE OF VARIABILITY

AIM:- Understating Statistical descriptions measure of central tendency(mean, median and mode) and measure of variability(variance and standard deviation).

THEORY:-

Measures of Central Tendency:

Mean: The mean, often referred to as the average, is the sum of all values in a dataset divided by the total number of values.

Formula: $\mu = \frac{\sum_{i=1}^{n} x_i}{n}$

Example: If you have a dataset of exam scores {85, 90, 92, 88, 95}, the mean score is $\frac{85+90+92+88+95}{5} = 90.$

Median: The median is the middle value of a dataset when it is ordered from least to greatest. If there is an even number of values, the median is the average of the two middle values.

Example: In the dataset {15, 20, 25, 30, 35}, the median is 25. In the dataset {10, 15, 20, 25, 30, 35}, the median is $\frac{20+25}{2} = 22.5.$

Mode: The mode is the value that appears most frequently in a dataset. A dataset can have one mode (unimodal), two modes (bimodal), or more than two modes (multimodal).
Example: In the dataset {10, 20, 20, 30, 40, 40, 40}, the mode is 40.

Measures of Variability:

Variance: Variance measures the spread of data points around the mean. It is calculated as the average of the squared differences between each data point and the mean.

Formula: $\sigma^2 = \frac{\sum_{i=1}^{n} (x_i - \mu)^2}{n}$

Example: For the dataset {10, 15, 20, 25, 30}, with a mean of 20, the variance is $\frac{(10-20)^2+(15-20)^2+(20-20)^2+(25-20)^2+(30-20)^2}{5} = 50.$

Standard Deviation: Standard deviation is the square root of the variance. It provides a measure of the average distance of data points from the mean.

Formula: $\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i-\mu)^2}{n}}$

Example: Using the same dataset as above, the standard deviation is $\sqrt{50} \approx 7.07$.

CODE:-

```
import statistics

data = [10, 15, 20, 25, 30, 40, 40, 40]

mean = statistics.mean(data)

print("Mean:", mean)

median = statistics.median(data)

print("Median:", median)

mode = statistics.mode(data)

print("Mode:", mode)

variance = statistics.variance(data)

print("Variance:", variance)

std_deviation = statistics.stdev(data)

print("Standard Deviation:", std_deviation)
```

OUTPUT:-

```
PS G:\New folder\AI MINI PROJECT3333333333> python -u "g:\New folder\AI MINI PROJECT3333333333\src\hello1.py"
Mean: 27.5
Median: 27.5
Mode: 40
Variance: 142.85714285714286
Standard Deviation: 11.952286093343936
PS G:\New folder\AI MINI PROJECT3333333333>
```

EXPERIMENT-03

DATA CLEANING

AIM:- Understating Data Cleaning(Treatment of outlier and missing values).

THEORY:-

Data cleaning is a crucial step in the data preprocessing pipeline, ensuring that the data is accurate, complete, and reliable for analysis. Here's an overview of two common tasks in data cleaning: treating outliers and handling missing values.

Treatment of Outliers: Outliers are data points that significantly deviate from the rest of the dataset. They can occur due to measurement errors, data entry mistakes, or genuine but rare events. Outliers can distort statistical analyses and machine learning models, hence it's important to handle them appropriately.

Methods to Treat Outliers:

Identifying Outliers: Visual inspection using box plots, scatter plots, or histograms, or statistical methods like Z-score or IQR (Interquartile Range) can help identify outliers.
Removing Outliers: If outliers are due to errors or do not represent genuine data points, they can be removed from the dataset. However, extreme caution must be exercised to ensure that valuable information is not lost.
Transforming Values: Transforming skewed data using logarithmic, square root, or Box-Cox transformations can reduce the impact of outliers.
Handling Missing Values: Missing values are common in real-world datasets due to various reasons like data entry errors, incomplete data collection, or data loss during transmission.

Methods to Handle Missing Values:

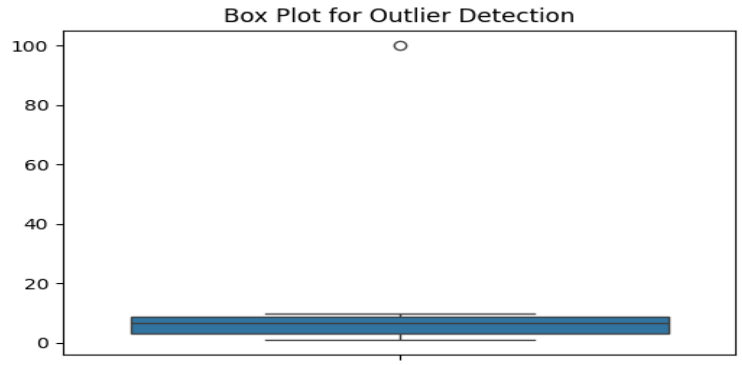
Deleting Missing Data: If missing values are few and randomly distributed, deleting rows or columns with missing values can be an option. However, this approach may lead to loss of valuable information.
Imputation: Missing values can be replaced with estimated values. Common imputation techniques include:
Mean/Median/Mode Imputation: Replace missing values with the mean, median, or mode of the column.
Forward Fill/Backward Fill: Use the last known value (forward fill) or next known value (backward fill) to fill missing values in time series data.
Interpolation: Use interpolation methods like linear interpolation or spline interpolation to estimate missing values based on neighboring data points.
Predictive Modeling: Build predictive models to estimate missing values based on other variables in the dataset.
Multiple Imputation: Generate multiple imputations for missing values and analyze each imputed dataset separately to incorporate uncertainty.

CODE:-

Box plot:-

```
import seaborn as sns
import matplotlib.pyplot as plt
data = [1, 2, 3, 4, 100, 6, 7, 8, 9, 10]
# Box plot for outlier detection
plt.figure(figsize=(6, 4))
sns.boxplot(data=data)
plt.title('Box Plot for Outlier Detection')
plt.show()
```

OUTPUT:-



z-scores :-

```
import numpy as np
data_with_outliers = [1, 2, 3, 4, 100, 6, 7, 8, 9, 1987]
# Function for outlier detection using Z-scores
def detect_outliers_zscore(data):
    threshold = 1  # Adjust threshold
    z_scores = np.abs((data - np.mean(data)) / np.std(data))
    outliers = [data[i] for i, z in enumerate(z_scores) if z > threshold]
    return outliers
outliers_zscore = detect_outliers_zscore(data_with_outliers)
print("Outliers detected using Z-scores:", outliers_zscore)
```

OUTPUT:-

```
PS G:\New folder\AI MINI PROJECT3333333333> python -u "g:\New folder\AI MINI PROJECT3333333333\src\hello 2.py"
Outliers detected using Z-scores: [1987]
PS G:\New folder\AI MINI PROJECT3333333333>
```

Median:-

```
import numpy as np
data = [1, 2, 3, 4, 100, 6, 7, 8, 9, 10]
def detect_outliers_median(data):
    median = np.median(data)
    mad = np.median([abs(x - median) for x in data])
    threshold = 3 * mad
    outliers = [x for x in data if abs(x - median) > threshold]
    return outliers
outliers_median = detect_outliers_median(data)
print("Outliers detected using median value:", outliers_median)
```

OUTPUT:-

```
PS G:\New folder\AI MINI PROJECT3333333333> python -u "g:\New folder\AI MINI PROJECT3333333333\src\hello 2.py"
Outliers detected using median value: [100]
PS G:\New folder\AI MINI PROJECT3333333333>
```

Missing value

```
import pandas as pd
# Sample DataFrame with missing values
data = {'A': [1, 2, 3, 4, None, 6, 7, None, 9, 10],
        'B': [100, 200, None, 400, 500, 600, 700, 800, None, 1000]}
df = pd.DataFrame(data)
# Detect missing values
missing_values = df.isnull().sum()
print("Missing values detected:")
print(missing_values)
```

OUTPUT:-

```
PS G:\New folder\AI MINI PROJECT3333333333> python -u "g:\New folder\AI MINI PROJECT3333333333\src\hello 2.py"
Missing values detected:
A    2
B    2
dtype: int64
PS G:\New folder\AI MINI PROJECT3333333333>
```

EXPERIMENT-04

DATA DISCRETIZATION AND DATA TRANSFORMATION

AIM:- Understating Data Discretization and Data Transformation in data mining.

THEORY:-

Data discretization is the process of transforming continuous variables into discrete intervals or categories. This is particularly useful when dealing with continuous data that needs to be represented in a more manageable form or when algorithms require categorical input.

Methods of Data Discretization:

Equal Width Binning: Divide the range of values into equal-width intervals. For example, if the range of values is 0 to 100 and we want 5 intervals, each interval will be 20 units wide (0-20, 21-40, etc.).
Equal Frequency Binning: Divide the data into intervals such that each interval contains roughly the same number of data points.
Clustering-Based Discretization: Use clustering algorithms to group similar data points into discrete intervals.
Entropy-Based Discretization: Partition the data into intervals based on the entropy of the target variable.

Data transformation involves altering the original data to make it more suitable for analysis. This can include scaling, normalization, or applying mathematical functions to transform the data distribution.

Methods of Data Transformation:

Normalization: Scale the data to have a mean of 0 and a standard deviation of 1, typically done on each feature independently.
Min-Max Scaling: Scale the data to a fixed range, usually between 0 and 1 or -1 and 1.
Log Transformation: Apply the natural logarithm to data to reduce skewness and make it more normally distributed.
Box-Cox Transformation: A family of power transformations that includes log transformation as a special case, but also works for negative values and can handle different transformations for different ranges of data.
PCA (Principal Component Analysis): A dimensionality reduction technique that transforms the original variables into a new set of uncorrelated variables called principal components.

CODE:-

Data transformation:-

```
import numpy as np
from sklearn.preprocessing import StandardScaler, MinMaxScaler, FunctionTransformer
from sklearn.compose import ColumnTransformer
import pandas as pd
data = np.array([[1, 2, 3], [4, 5, 6],  [7, 8, 9]])
# Convert data to pandas DataFrame for better visualization
df = pd.DataFrame(data, columns=['Feature 1', 'Feature 2', 'Feature 3'])
# Standardization (Z-score normalization)
scaler = StandardScaler()
standardized_data = scaler.fit_transform(data)
print("Standardized data:\n", standardized_data)
# Min-Max Scaling
min_max_scaler = MinMaxScaler()
min_max_scaled_data = min_max_scaler.fit_transform(data)
print("\nMin-Max scaled data:\n", min_max_scaled_data)

# Log Transformation

log_transformer = FunctionTransformer(np.log1p, validate=True)
log_transformed_data = log_transformer.fit_transform(data)
print("\nLog-transformed data:\n", log_transformed_data)

# Box-Cox Transformation

boxcox_transformer = FunctionTransformer(lambda x: np.squeeze(np.c_[np.array(x)]), validate=True)
boxcox_transformed_data = boxcox_transformer.fit_transform(data)
print("\nBox-Cox transformed data:\n", boxcox_transformed_data)

# Applying transformations to specific columns

column_transformer = ColumnTransformer(transformers=[
    ('log', FunctionTransformer(np.log1p, validate=True), ['Feature 1']),
    ('min_max', MinMaxScaler(), ['Feature 2']),
    ('standard', StandardScaler(), ['Feature 3'])
])
transformed_data = column_transformer.fit_transform(df)

print("\nTransformed data:\n", transformed_data)
```

OUTPUT:-

```
PS G:\New folder\AI MINI PROJECT3333333333> python -u "g:\New folder\AI MINI PROJECT3333333333\src\hello 2.py"
Standardized data:
 [[-1.22474487 -1.22474487 -1.22474487]
 [ 0.          0.          0.        ]
 [ 1.22474487  1.22474487  1.22474487]]

Min-Max scaled data:
 [[0.  0.  0. ]
 [0.5 0.5 0.5]
 [1.  1.  1. ]]

Log-transformed data:
 [[0.69314718 1.09861229 1.38629436]
 [1.60943791 1.79175947 1.94591015]
 [2.07944154 2.19722458 2.30258509]]

Box-Cox transformed data:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]

Transformed data:
 [[ 0.69314718  0.         -1.22474487]
 [ 1.60943791  0.5         0.        ]
 [ 2.07944154  1.          1.22474487]]
PS G:\New folder\AI MINI PROJECT3333333333>
```

Data discretization

```
import pandas as pd
data = {'Age': [22, 35, 45, 28, 32, 50, 28, 40, 60, 70],
        'Income': [30000, 40000, 60000, 80000, 120000, 150000, 100000, 80000, 110000, 100000]}
# Convert data to pandas DataFrame
df = pd.DataFrame(data)
# Equal Width Binning for Age
age_bins = pd.cut(df['Age'], bins=3, labels=['Young', 'Middle-aged', 'Senior'])
# Equal Frequency Binning for Income
income_bins = pd.qcut(df['Income'], q=3, labels=['Low', 'Medium', 'High'])
# Add discretized columns to DataFrame
df['Age Group'] = age_bins
df['Income Group'] = income_bins
print("Data with discretized columns:")
print(df)
```

OUTPUT:-

```
PS G:\New folder\AI MINI PROJECT3333333333> python -u "g:\New folder\AI MINI PROJECT3333333333\src\hello 2.py"
Data with discretized columns:
   Age  Income    Age Group Income Group
0   22   30000        Young          Low
1   35   40000        Young          Low
2   45   60000  Middle-aged          Low
3   28   80000        Young          Low
4   32  120000        Young         High
5   50  150000  Middle-aged         High
6   28  100000        Young       Medium
7   40   80000  Middle-aged          Low
8   60  110000       Senior         High
9   70  100000       Senior       Medium
PS G:\New folder\AI MINI PROJECT3333333333>
```

EXPERIMENT-05

CORRELATIONS

AIM:- Correlations between variables using heat map.

THEORY:-

Correlation: Correlation measures the strength and direction of the linear relationship between two variables.

It ranges from -1 to 1, where:

1 indicates a perfect positive correlation (as one variable increases, the other increases).

-1 indicates a perfect negative correlation (as one variable increases, the other decreases).

0 indicates no linear correlation between the variables.

Correlation analysis helps understand how variables are related and whether changes in one variable are associated with changes in another.

Heatmap: A heatmap is a graphical representation of data where values in a matrix are represented as colors.

It is commonly used to visualize the correlation matrix, especially in exploratory data analysis.

In correlation heatmaps:

Rows and columns represent variables.

Each cell's color represents the correlation coefficient between the corresponding pair of variables.

Colors range from cool colors (e.g., blue) for negative correlations to warm colors (e.g., red) for positive correlations.

Intensity of colors indicates the strength of the correlation: darker shades represent stronger correlations.

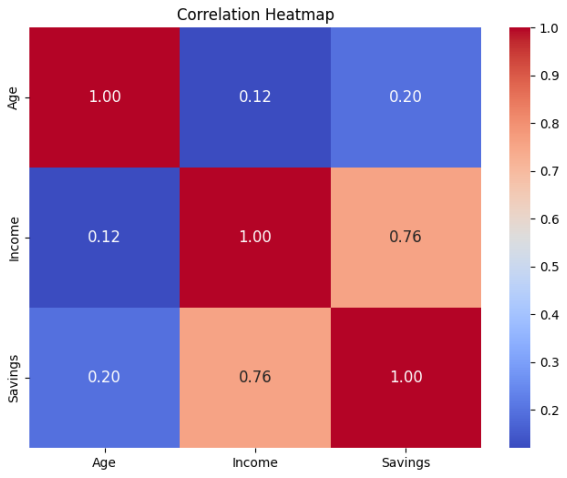
Heatmaps make it easy to identify patterns and relationships in the data, especially when dealing with a large number of variables.

They provide a visual summary of the correlations between variables, helping researchers and analysts gain insights quickly and intuitively.

CODE:-

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
data = { 'Age': [22, 35, 45, 28, 32],
    'Income': [30000, 40000, 60000, 80000, 120000],
    'Savings': [5000, 8000, 10000, 2000, 30000] }
# Create DataFrame
df = pd.DataFrame(data)
# Calculate correlation matrix
corr_matrix = df.corr()
# Plot heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f", annot_kws={"size": 12})
plt.title('Correlation Heatmap')
plt.show()
```

OUTPUT:-



EXPERIMENT-06

APRIORI ALGORITHM

AIM:- Implemention of Apriori algorithm for frequent pattern mining.

THEORY:-

The Apriori algorithm is a classic algorithm for frequent itemset mining and association rule learning in transactional databases. It was proposed by Rakesh Agrawal and Ramakrishnan Srikant in 1994. The algorithm is widely used in market basket analysis, recommendation systems, and other applications involving discovering interesting relationships between items in large datasets.

Support:
Support measures the frequency of occurrence of an itemset in a dataset.
For an itemset X, support is defined as the proportion of transactions in the dataset that contain all the items in X.
A frequent itemset is one whose support is greater than or equal to a specified minimum support threshold.

Apriori Principle:
The Apriori principle states that if an itemset is frequent, then all of its subsets must also be frequent.
This principle helps in reducing the search space during frequent itemset generation. If an itemset is found to be infrequent, its supersets need not be explored further.

Frequent Itemset Generation:
The Apriori algorithm generates frequent itemsets by iteratively increasing the size of itemsets and pruning those that are infrequent.
It starts by finding all frequent 1-itemsets (single items) by scanning the dataset once and counting the support of each item.
Then, it iteratively generates frequent itemsets of size k (k > 1) by joining frequent (k-1)-itemsets and pruning those that have infrequent subsets.
This process continues until no new frequent itemsets can be found.

Association Rule Generation:
Once frequent itemsets are identified, association rules are generated from them based on user-specified metrics such as confidence, lift, or conviction.
An association rule is an implication of the form X -> Y, where X and Y are itemsets, and $X \cap Y = \emptyset$ (X and Y are disjoint).

The confidence of a rule X -> Y measures the conditional probability of Y given X, and it is calculated as the support of ($X \cup Y$) divided by the support of X.

CODE:-

```
import pandas as pd
from mlxtend.frequent_patterns import apriori
from mlxtend.frequent_patterns import association_rules
# Sample transactions
data = {'TID': [1, 2, 3, 4, 5],
        'Items': [['bread', 'milk', 'butter'],
                  ['bread', 'milk', 'butter', 'beer'],
                  ['bread', 'milk'],
                  ['bread', 'butter'],
                  ['bread', 'milk', 'butter', 'beer']]}
df = pd.DataFrame(data)
# Convert list of items into a one-hot encoded DataFrame
oht = df['Items'].str.join('|').str.get_dummies()
oht = oht.rename(columns=lambda x: x.strip())
# Apply Apriori algorithm
frequent_itemsets = apriori(oht, min_support=0.5, use_colnames=True)
# Apply association rule mining
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.5)
print("Frequent Itemsets:")
print(frequent_itemsets)
print("\nAssociation Rules:")
print(rules)
```

OUTPUT:-

```
PS G:\New folder\AI MINI PROJECT3333333333> python -u "g:\New folder\AI MINI PROJECT3333333333\src\hello 2.py"
C:\Users\Abc1\AppData\Local\Programs\Python\Python310\lib\site-packages\mlxtend\frequent_patterns\fpcommon.py:109: DeprecationWarni
g: DataFrames with non-bool types result in worse computationalperformance and their support might be discontinued in the future.Pl
ase use a DataFrame with bool type
  warnings.warn(
Frequent Itemsets:
   support              itemsets
0      1.0               (bread)
1      0.8              (butter)
2      0.8                (milk)
3      0.8        (butter, bread)
4      0.8          (milk, bread)
5      0.6         (milk, butter)
6      0.6  (butter, milk, bread)

Association Rules:
        antecedents      consequents  antecedent support  consequent support  ...    lift  leverage  conviction  zhangs_metric
0          (butter)          (bread)                 0.8                 1.0  ...  1.0000      0.00         inf           0.00
1           (bread)         (butter)                 1.0                 0.8  ...  1.0000      0.00         1.0           0.00
2            (milk)          (bread)                 0.8                 1.0  ...  1.0000      0.00         inf           0.00
3           (bread)           (milk)                 1.0                 0.8  ...  1.0000      0.00         1.0           0.00
4            (milk)         (butter)                 0.8                 0.8  ...  0.9375     -0.04         0.8          -0.25
5          (butter)           (milk)                 0.8                 0.8  ...  0.9375     -0.04         0.8          -0.25
6    (milk, butter)          (bread)                 0.6                 1.0  ...  1.0000      0.00         inf           0.00
7   (bread, butter)           (milk)                 0.8                 0.8  ...  0.9375     -0.04         0.8          -0.25
8     (milk, bread)         (butter)                 0.8                 0.8  ...  0.9375     -0.04         0.8          -0.25
9          (butter)    (milk, bread)                 0.8                 0.8  ...  0.9375     -0.04         0.8          -0.25
10           (milk)  (bread, butter)                 0.8                 0.8  ...  0.9375     -0.04         0.8          -0.25
11          (bread)   (milk, butter)                 1.0                 0.6  ...  1.0000      0.00         1.0           0.00

[12 rows x 10 columns]
PS G:\New folder\AI MINI PROJECT3333333333>
```

EXPERIMENT-07

BAYES CALSSIFICATION METHOD

AIM:- Implementation of Bayes Classification Method.

THEORY:-

Bayesian classification, also known as Naive Bayes classification, is a probabilistic classification method based on Bayes' theorem. It's called "naive" because it makes the assumption of feature independence within each class, which means that the presence of a particular feature in a class is independent of the presence of any other feature.

Bayes' Theorem:
Bayes' theorem is a fundamental theorem in probability theory that describes the probability of an event based on prior knowledge of conditions that might be related to the event. Mathematically, it is expressed as:

$$P(C_k|X) = \frac{P(X|C_k) \times P(C_k)}{P(X)}$$

Where:

- $P(C_k|X)$ is the posterior probability of class C_k given feature vector X.
- $P(X|C_k)$ is the likelihood of observing feature vector X given class C_k.
- $P(C_k)$ is the prior probability of class C_k.
- $P(X)$ is the probability of observing feature vector X (the evidence).

Naive Bayes Classifier:
In the context of classification, Bayes' theorem is used to predict the probability of a given sample belonging to each class.
The "naive" assumption of feature independence simplifies the computation by assuming that the presence of a feature in a class is independent of the presence of any other feature. This allows us to factorize the likelihood term as the product of individual feature probabilities.
The Naive Bayes classifier predicts the class with the highest posterior probability for a given sample.

Types of Naive Bayes Classifiers:

Gaussian Naive Bayes: Assumes that continuous features follow a Gaussian distribution.
Multinomial Naive Bayes: Suitable for features that represent counts or frequencies (e.g., word counts in text classification).
Bernoulli Naive Bayes: Applicable when features are binary variables (e.g., presence or absence of a feature).
Complementary Naive Bayes: Designed to address class imbalance by giving less weight to the dominant class.

CODE:-

```
import numpy as np
class NaiveBayesClassifier:
    def __init__(self):
        self.class_prior = None
        self.class_conditional_probs = None
    def fit(self, X, y):
        self.classes = np.unique(y)
        self.class_prior = {c: np.mean(y == c) for c in self.classes}
        self.class_conditional_probs = {}
        for c in self.classes:
            class_indices = np.where(y == c)
            class_data = X[class_indices]
            self.class_conditional_probs[c] = {
                'mean': np.mean(class_data, axis=0),
                'std': np.std(class_data, axis=0) }
    def predict(self, X):
        predictions = []
        for sample in X:
            probabilities = []
            for c in self.classes:
                prior = self.class_prior[c]
                likelihood = np.prod(self._gaussian_prob(sample,
self.class_conditional_probs[c]['mean'], self.class_conditional_probs[c]['std']))
                probabilities.append(prior * likelihood)
            predictions.append(self.classes[np.argmax(probabilities)])
        return predictions
    def _gaussian_prob(self, x, mean, std):
        return 1 / (np.sqrt(2 * np.pi) * std) * np.exp(-0.5 * ((x - mean) / std) ** 2)

# Example usage
X = np.array([[1, 1], [2, 3], [3, 4], [4, 6]])
y = np.array([0, 0, 1, 1])
classifier = NaiveBayesClassifier()
classifier.fit(X, y)
new_samples = np.array([[1, 2], [3, 5]])
predictions = classifier.predict(new_samples)
print("Predictions:", predictions)
```

OUTPUT:-

```
PS G:\New folder\AI MINI PROJECT3333333333> python -u "g:\New folder\AI MINI PROJECT3333333333\src\hello 2.py"
Predictions: [0, 1]
PS G:\New folder\AI MINI PROJECT3333333333>
```

EXPERIMENT-08

k- NEAREST -NEIGHBORS CLASSIFIERS

AIM:- Implemention of k-Nearesr-Neighbors Classifiers.

THEORY:-

The k-Nearest Neighbors (k-NN) algorithm is a simple and intuitive classification algorithm that works based on the distance between data points.
It belongs to the family of instance-based or lazy learning algorithms, as it doesn't explicitly learn a model during training. Instead, it memorizes the training data and makes predictions based on the similarity of new instances to known instances.

Algorithm:

During training, the k-NN algorithm simply stores the feature vectors and corresponding class labels of the training data.
To make predictions for a new instance, the algorithm computes the distances between the new instance and all instances in the training data.
It then selects the k nearest neighbors (i.e., the k instances with the smallest distances) to the new instance.
Finally, it assigns the class label to the new instance based on a majority vote among the labels of its k nearest neighbors.

Distance Metric:

The choice of distance metric (e.g., Euclidean distance, Manhattan distance, etc.) plays a crucial role in determining the similarity between instances.
The most commonly used distance metric is the Euclidean distance, which calculates the straight-line distance between two points in Euclidean space.

Hyperparameter k:

The value of k is a hyperparameter that needs to be specified before training the model.
The selection of k can have a significant impact on the performance of the algorithm.
Smaller values of k tend to capture more local patterns in the data but may be sensitive to noise and outliers. Larger values of k provide smoother decision boundaries but may lead to misclassification of instances near the decision boundary.

CODE :-

```
import numpy as np
class KNNClassifier:
    def __init__(self, k=8):
        self.k = k
    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train
    def predict(self, X_test):
        predictions = []
        for sample in X_test:
            distances = [np.linalg.norm(sample - x) for x in self.X_train]
            nearest_indices = np.argsort(distances)[:self.k]
            nearest_labels = [self.y_train[i] for i in nearest_indices]
            prediction = max(set(nearest_labels), key=nearest_labels.count)
            predictions.append(prediction)
        return predictions

# Example usage
X_train = np.array([[1, 9], [2, 3], [3, 4], [4, 5], [5, 6]])
y_train = np.array([0, 0, 1, 1, 1])
X_test = np.array([[2, 2], [4, 4]])
# Create and train the classifier
classifier = KNNClassifier(k=3)
classifier.fit(X_train, y_train)
# Predict classes for new samples
predictions = classifier.predict(X_test)
print("Predictions:", predictions)
```

OUTPUT:-

```
PS G:\New folder\AI MINI PROJECT3333333333> python -u "g:\New folder\AI MINI PROJECT3333333333\src\hello 2.py"
Predictions: [1, 1]
PS G:\New folder\AI MINI PROJECT3333333333>
```

EXPERIMENT-09

k- MEANS CLUSTERING

AIM:- Implemention of k-Means Clustering.

THEORY:-

k-Means clustering is a popular unsupervised learning algorithm used for partitioning a dataset into a predefined number of clusters.
It aims to group similar data points together and discover underlying patterns or structure in the data.

Algorithm:

Given a dataset with n data points and a predefined number of clusters k, the k-Means algorithm works as follows:
Randomly initialize k cluster centroids (i.e., the centers of the clusters).
Assign each data point to the nearest centroid based on some distance metric (typically Euclidean distance).
Update the centroids by computing the mean of the data points assigned to each cluster.
Repeat steps 2 and 3 until convergence, where convergence occurs when the centroids no longer change significantly or a maximum number of iterations is reached.

Objective Function:

The k-Means algorithm minimizes the within-cluster sum of squares (WCSS), also known as the inertia or distortion.
The WCSS is defined as the sum of the squared distances between each data point and its assigned centroid.
Minimizing the WCSS encourages tight clusters with data points close to their centroids.

Initialization:

The choice of initial cluster centroids can significantly impact the convergence and quality of the clustering.
Common initialization methods include random initialization, k-Means++, and selecting data points as initial centroids based on some criteria.

Number of Clusters (k):

The number of clusters k is a hyperparameter that needs to be specified before training the model.
Choosing the optimal value of k is often challenging and may require domain knowledge or techniques such as the elbow method or silhouette analysis.

CODE :-

```
import numpy as np
class KMeans:
    def __init__(self, n_clusters, max_iter=300):
        self.n_clusters = n_clusters
        self.max_iter = max_iter
    def fit(self, X):
        # Randomly initialize centroids
        centroids = X[np.random.choice(X.shape[0], self.n_clusters, replace=False)]
        for _ in range(self.max_iter):
            # Assign each data point to the nearest centroid
            labels = np.argmin(np.linalg.norm(X[:, np.newaxis] - centroids, axis=2), axis=1)
            # Update centroids based on the mean of data points assigned to each cluster
            new_centroids = np.array([X[labels == i].mean(axis=0) for i in range(self.n_clusters)])
            # Check for convergence
            if np.allclose(centroids, new_centroids):
                break
            centroids = new_centroids
        self.labels_ = labels
        self.cluster_centers_ = centroids



# Example usage
X = np.array([[1, 2], [1, 4], [1, 0],
              [4, 2], [4, 4], [4, 0]])
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
print("Cluster labels:", kmeans.labels_)
print("Cluster centers:", kmeans.cluster_centers_)
```

OUTPUT:-

```
PS G:\New folder\AI MINI PROJECT3333333333> python -u "g:\New folder\AI MINI PROJECT3333333333\src\hello 2.py"
Cluster labels: [1 1 1 0 0 0]
Cluster centers: [[4 2]
 [1 2]]
PS G:\New folder\AI MINI PROJECT3333333333>
```

EXPERIMENT-10

HIERARCHICAL METHODS

AIM:- Implemention Hierarchical Methods.

THEORY:-

Hierarchical clustering organizes data points into a hierarchy of clusters. In agglomerative hierarchical clustering, each data point starts as its own cluster and is gradually merged with nearby clusters. The process continues until all data points belong to a single cluster or a desired number of clusters is reached.

The algorithm relies on a distance metric to measure similarity between data points or clusters. Common linkage criteria, like single, complete, average, or Ward's linkage, determine how the distance between clusters is calculated during merging.

A dendrogram visually represents the clustering process, showing the arrangement of clusters at different levels of granularity. Hierarchical clustering is versatile, capturing hierarchical structures and providing insights into relationships between clusters. However, it can be computationally expensive and sensitive to the choice of distance metric and linkage criteria.

CODE:-

```
import numpy as np
class AgglomerativeClustering:
    def __init__(self, n_clusters):
        self.n_clusters = n_clusters
    def fit(self, X):
        self.labels_ = np.arange(len(X))  # Initialize each data point as its own cluster
        self.n_clusters_ = len(X)
        while self.n_clusters_ > self.n_clusters:
            min_distance = np.inf
            merge_indices = None
            for i in range(len(X)):
                for j in range(i + 1, len(X)):
                    distance = np.linalg.norm(X[i] - X[j])
                    if distance < min_distance:
                        min_distance = distance
                        merge_indices = (i, j)
```

```
            # Merge the two clusters with the smallest distance
            cluster1, cluster2 = merge_indices
            self.labels_[self.labels_ == cluster2] = cluster1
            self.n_clusters_ -= 1
        # Renumber clusters to ensure sequential numbering
        unique_labels = np.unique(self.labels_)
        for i, label in enumerate(unique_labels):
            self.labels_[self.labels_ == label] = i


# Example usage
X = np.array([[1, 2], [1, 4], [1, 0],
              [4, 2], [4, 4], [4, 0]])
agglomerative = AgglomerativeClustering(n_clusters=2)
agglomerative.fit(X)
print("Cluster labels:", agglomerative.labels_)
```

OUTPUT:-

```
PS G:\New folder\AI MINI PROJECT3333333333> python -u "g:\New folder\AI MINI PROJECT3333333333\src\hello 2.py"
Cluster labels: [0 0 1 2 3 4]
PS G:\New folder\AI MINI PROJECT3333333333>
```

EXPERIMENT-11

DENSITY-BASED METHODS

AIM:- Implementation Density-Based Methods(DBSCAN).

THEORY:-

DBSCAN is a density-based clustering algorithm that partitions a dataset into clusters of varying shapes and sizes based on the density of data points.
Unlike partitioning methods like k-means, DBSCAN does not require specifying the number of clusters in advance and can identify outliers as noise.

Core Concepts:

Epsilon (ε) Neighborhood: For each data point, DBSCAN defines a neighborhood around it within a radius ε.
Core Point: A data point is considered a core point if it has at least a minimum number of neighbors (min_samples) within its ε-neighborhood.
Border Point: A data point is considered a border point if it is reachable from a core point but does not have enough neighbors to be considered a core point itself.
Noise Point (Outlier): A data point is considered noise if it is neither a core point nor a border point.

Algorithm Steps:

Step 1: Neighborhood Query: For each data point, DBSCAN identifies its ε-neighborhood.
Step 2: Core Point Identification: Data points with at least min_samples neighbors within their ε-neighborhood are considered core points.
Step 3: Cluster Formation: DBSCAN forms clusters by assigning each core point and its reachable neighbors to the same cluster. Border points are assigned to the cluster of their corresponding core point.
Step 4: Noise Handling: Data points that are not core points or border points are marked as noise or outliers.

Key Parameters:

ε (epsilon): The maximum radius of the neighborhood around each data point.
min_samples: The minimum number of data points required to form a dense region (core point).

CODE-

```
import numpy as np
class DBSCAN:
    def __init__(self, eps=0.5, min_samples=5):
        self.eps = eps
        self.min_samples = min_samples
    def fit(self, X):
        self.X = X
        self.labels_ = np.zeros(len(X), dtype=int)
        self.cluster_id = 0
        for i in range(len(X)):
            if self.labels_[i] != 0:
                continue
            neighbors = self.region_query(i)
            if len(neighbors) < self.min_samples:
                self.labels_[i] = -1  # Mark as noise
            else:
                self.cluster_id += 1
                self.grow_cluster(i, neighbors, self.cluster_id)
    def region_query(self, i):
        neighbors = []
        for j in range(len(self.X)):
            if np.linalg.norm(self.X[i] - self.X[j]) < self.eps:
                neighbors.append(j)
        return neighbors
    def grow_cluster(self, i, neighbors, cluster_id):
        self.labels_[i] = cluster_id
        while len(neighbors) > 0:
            j = neighbors.pop()
            if self.labels_[j] == 0:
                self.labels_[j] = cluster_id
                new_neighbors = self.region_query(j)
                if len(new_neighbors) >= self.min_samples:
                    neighbors.extend(new_neighbors)
# Example usage
X = np.array([[1, 2], [1, 4], [1, 0],
              [4, 2], [4, 4], [4, 0]])
dbscan = DBSCAN(eps=1, min_samples=2)
dbscan.fit(X)
print("Cluster labels:", dbscan.labels_)
```

OUTPUT:-

```
PS G:\New folder\AI MINI PROJECT3333333333> python -u "g:\New folder\AI MINI PROJECT3333333333\src\hello 2.py"
Cluster labels: [-1 -1 -1 -1 -1 -1]
PS G:\New folder\AI MINI PROJECT3333333333>
```