VIGNAN'S
FOUNDATION FOR SCIENCE, TECHNOLOGY & RESEARCH
(Deemed to be University) - Estd. u/s 3 of UGC Act 1956

DEPARTMENT OF INFORMATION TECHNOLOGY
22IT307 Machine Learning, III B.TECH I Semester (2025-26)
Lab Manual
****

List of Experiments

S. No	**Name of the Experiment**
1	Implement data cleaning techniques on a real-world dataset
2	Implement normalization technique on a real-world dataset
3	Apply dimensionality reduction techniques (e.g., PCA) to a dataset and analyze the results
4	Implement K-Means clustering algorithm on a dataset
5	Implement Hierarchical clustering algorithm on a dataset
6	Implement Density-based clustering algorithm on a dataset
7	Implement Apriori algorithm
8	Implement FP Growth algorithm
9	Implement Decision Tree classification algorithm on a labeled dataset
10	Implement k-Nearest Neighbors (k-NN) classification algorithm on a labeled dataset
11	Implement Support Vector Machines (SVM) classification algorithm on a labeled dataset
12	Implement Naïve Bayes classification algorithm on a labeled dataset
13	Build and evaluate a linear regression model to predict a continuous target variable
14	Build and evaluate a polynomial regression model

1. Implement Data Cleaning Techniques on a Real-World Dataset

Objective

To understand and apply data cleaning techniques to preprocess a real-world dataset for analysis and model building.

Prerequisites

- Python programming.
- Libraries: pandas, numpy, matplotlib, seaborn.

Dataset Description

- **Dataset Used**: Titanic dataset.
- **Source**: Available on Kaggle.
- **Features**:
    - PassengerId: Unique ID for each passenger.
    - Survived: Survival status (0 = No, 1 = Yes).
    - Pclass: Passenger class (1st, 2nd, 3rd).
    - Name: Passenger name.
    - Age: Age of the passenger.
    - Fare: Ticket fare.
    - Embarked: Port of embarkation (C = Cherbourg, Q = Queenstown, S = Southampton).

Procedure

Step 1: Load the Dataset

```
import pandas as pd

# Load the dataset

url = "https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv"

df = pd.read_csv(url)

# View the first few rows

print(df.head())
```

Step 2: Explore the Dataset

```
# Basic information about the dataset
print(df.info())

# Summary statistics
print(df.describe())

# Check for missing values
print(df.isnull().sum())

# Check for duplicates
print(df.duplicated().sum())
```

Step 3: Handle Missing Values

```
# Fill missing 'Age' with median
df['Age'] = df['Age'].fillna(df['Age'].median())

# Fill missing 'Embarked' with mode
df['Embarked'] = df['Embarked'].fillna(df['Embarked'].mode()[0])

# Drop rows with missing 'Cabin' data (if applicable)
df.drop('Cabin', axis=1, inplace=True)
```

Step 4: Handle Duplicates

```
# Remove duplicate rows
df = df.drop_duplicates()

print(f"Duplicates removed: {df.duplicated().sum()}")
```

Step 5: Correct Data Types

```
# Convert 'Survived' to category
df['Survived'] = df['Survived'].astype('category')
# Convert 'Pclass' to category
df['Pclass'] = df['Pclass'].astype('category')
```

Step 6: Handle Outliers

```
import seaborn as sns
import matplotlib.pyplot as plt
# Boxplot to detect outliers in 'Fare'
sns.boxplot(x=df['Fare'])
plt.show()
# Remove outliers (e.g., using the IQR method)
Q1 = df['Fare'].quantile(0.25)
Q3 = df['Fare'].quantile(0.75)
IQR = Q3 - Q1
# Define outlier boundaries
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
# Remove outliers
df = df[(df['Fare'] >= lower_bound) & (df['Fare'] <= upper_bound)]
```

Step 7: Normalize Numeric Columns

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
# Normalize 'Age' and 'Fare'
df[['Age', 'Fare']] = scaler.fit_transform(df[['Age', 'Fare']])
print(df[['Age', 'Fare']].head())
```

Step 8: Encode Categorical Variables

```
# One-hot encode 'Embarked'
df = pd.get_dummies(df, columns=['Embarked'], drop_first=True)
print(df.head())
```

Results

- The dataset is cleaned with missing values filled, duplicates removed, and data normalized.
- Key numeric variables (Age, Fare) are scaled.
- Categorical variables (Embarked) are encoded for analysis.

Viva Questions

1. Why is handling missing data important in data cleaning?
2. What is the impact of outliers on machine learning models?
3. Explain the difference between Min-Max normalization and Standardization.

2. Implement Normalization Technique on a Real-World Dataset

Objective

To apply normalization techniques to scale the features of a real-world dataset for better performance in machine learning models.

Prerequisites

- Python programming.
- Libraries: pandas, numpy, sklearn.

Dataset Description

Dataset Used: Iris dataset.
Source: Available in the sklearn library.
Features:
- Sepal Length
- Sepal Width
- Petal Length
- Petal Width
- Target: Species of Iris (Setosa, Versicolor, Virginica).

Procedure

Step 1

Load the Dataset:

```
from sklearn.datasets import load_iris
import pandas as pd

# Load dataset
iris = load_iris()

# Create a DataFrame
data = pd.DataFrame(iris.data, columns=iris.feature_names)
data['target'] = iris.target

# View the first few rows
print(data.head())
```

Step 2

Explore the Dataset:

```
# Check data types and missing values
print(data.info())
```

```
# Summary statistics
print(data.describe())
```

Step 3

Apply Normalization:

```
from sklearn.preprocessing import MinMaxScaler

# Initialize scaler
scaler = MinMaxScaler()

# Normalize numeric features
normalized_data = scaler.fit_transform(data.iloc[:, :-1])

# Convert to DataFrame
normalized_df = pd.DataFrame(normalized_data, columns=iris.feature_names)

# Add target column back
normalized_df['target'] = data['target']

print(normalized_df.head())
```

Step 4

Visualize the Normalized Data:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Pairplot to visualize relationships
sns.pairplot(normalized_df, hue='target', diag_kind='kde')
plt.show()
```

Results

The dataset has been normalized, ensuring all numeric features lie within the range [0, 1]. This helps improve the performance and convergence rate of machine learning models.

Conclusion

Normalization ensures that all features contribute equally to the model training process by scaling them to a similar range. This is particularly important for distance-based algorithms.

Viva Questions

1. What is the difference between normalization and standardization?
2. Why is normalization important for distance-based algorithms like k-NN or SVM?
3. How does Min-Max normalization work?

3. Apply Dimensionality Reduction Techniques (e.g., PCA) to a Dataset and Analyze the Results

Objective

To reduce the dimensionality of a dataset using Principal Component Analysis (PCA) and analyze the transformed dataset.

Prerequisites

- Python programming.
- Libraries: pandas, numpy, sklearn, matplotlib, seaborn.

Dataset Description

Dataset Used: Breast Cancer dataset.
Source: Available in the sklearn library.
Features:
- Various diagnostic measurements of breast tumors.
- Target: Benign (0) or Malignant (1).

Procedure

Step 1

Load the Dataset:

```
from sklearn.datasets import load_breast_cancer
import pandas as pd

# Load dataset
cancer = load_breast_cancer()

# Create a DataFrame
data = pd.DataFrame(cancer.data, columns=cancer.feature_names)
data['target'] = cancer.target

# View the first few rows
print(data.head())
```

Step 2

Explore the Dataset:

```
# Basic information
print(data.info())

# Summary statistics
print(data.describe())
```

Step 3

Apply PCA:

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Standardize the data
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data.iloc[:, :-1])

# Apply PCA
pca = PCA(n_components=2)
principal_components = pca.fit_transform(scaled_data)

# Create a DataFrame with principal components
pca_df = pd.DataFrame(data=principal_components, columns=['PC1', 'PC2'])

# Add target column
pca_df['target'] = data['target']

print(pca_df.head())
```

Step 4

Analyze the Results:

```
# Explained variance ratio
print(f'Explained Variance Ratio: {pca.explained_variance_ratio_}')
```

Step 5

Visualize the PCA Results:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Scatter plot of the principal components
sns.scatterplot(data=pca_df, x='PC1', y='PC2', hue='target', palette='viridis')
plt.title('PCA - Breast Cancer Dataset')
plt.show()
```

Results

The dataset dimensionality was successfully reduced to two principal components, explaining a significant proportion of the variance. The transformed dataset was visualized to highlight patterns and separability of the classes.

Conclusion

PCA helps in reducing the dimensionality of a dataset by retaining the most important features that explain the majority of the variance. It aids in visualizing high-dimensional data and reducing computational complexity.

Viva Questions

1. What is the purpose of dimensionality reduction?
2. How does PCA differ from feature selection?
3. What is the role of standardization in PCA?

4. Implement K-Means Clustering Algorithm on a Dataset

Objective

To apply the K-Means clustering algorithm on a dataset to identify clusters within the data and analyze the results.

Prerequisites

- Python programming.
- Libraries: pandas, numpy, sklearn, matplotlib, seaborn.

Dataset Description

- **Dataset Used:** Iris dataset.
- **Source:** Available in the sklearn library.
- **Features:**
    - Sepal length
    - Sepal width
    - Petal length
    - Petal width
- **Target:** Species of Iris flowers (Setosa, Versicolor, Virginica).

Procedure

Step 1: Load the Dataset

```
from sklearn.datasets import load_iris

import pandas as pd


# Load dataset

iris = load_iris()


# Create a DataFrame

data = pd.DataFrame(iris.data, columns=iris.feature_names)

data['target'] = iris.target


# View the first few rows

print(data.head())
```

Step 2: Explore the Dataset

```
# Basic information
print(data.info())


# Summary statistics
print(data.describe())
```

Step 3: Apply K-Means Clustering

```
from sklearn.cluster import KMeans


# Define the model with the number of clusters (k=3 for Iris dataset)
kmeans = KMeans(n_clusters=3, random_state=42)


# Fit the model to the data
kmeans.fit(data.iloc[:, :-1])


# Assign the cluster labels to the dataset
data['cluster'] = kmeans.labels_


# View the first few rows with cluster labels
print(data.head())
```

Step 4: Analyze the Results

```
# Cluster centers
print("Cluster Centers: \n", kmeans.cluster_centers_)


# Inertia (sum of squared distances to cluster centers)
print(f"Inertia: {kmeans.inertia_}")
```

Step 5: Visualize the Clustering Results

```
import seaborn as sns
import matplotlib.pyplot as plt


# Scatter plot of Sepal length vs. Sepal width, colored by cluster
sns.scatterplot(data=data, x='sepal length (cm)', y='sepal width (cm)', hue='cluster',
palette='viridis')
plt.title('K-Means Clustering - Iris Dataset')
plt.show()
```

Results

The dataset was successfully clustered into three groups using K-Means. The cluster centers and inertia values were calculated. The visualization helped to observe the separation of the clusters based on Sepal length and Sepal width.

Conclusion

K-Means clustering is effective in identifying groups within the dataset, especially when the number of clusters (k) is known or can be determined through techniques like the elbow method. The clustering results can be analyzed through the center points and inertia to assess the quality of the model.

Viva Questions

1. What is the K-Means algorithm, and how does it work?
2. How do you determine the optimal value of k in K-Means?
3. What is inertia in K-Means, and what does it indicate?
4. What are the limitations of K-Means clustering?

5. Implement Hierarchical Clustering Algorithm on a Dataset

Objective

To apply the Hierarchical Clustering algorithm to a dataset and visualize the results to identify clusters.

Prerequisites

- Python programming.
- Libraries: pandas, numpy, sklearn, matplotlib, seaborn, scipy.

Dataset Description

- **Dataset Used:** Iris dataset.
- **Source:** Available in the sklearn library.
- **Features:**
  - Sepal length
  - Sepal width
  - Petal length
  - Petal width
- **Target:** Species of Iris flowers (Setosa, Versicolor, Virginica).

Procedure

Step 1: Load the Dataset

```
from sklearn.datasets import load_iris

import pandas as pd


# Load dataset

iris = load_iris()


# Create a DataFrame

data = pd.DataFrame(iris.data, columns=iris.feature_names)

data['target'] = iris.target


# View the first few rows

print(data.head())
```

Step 2: Explore the Dataset

```
# Basic information
print(data.info())

# Summary statistics
print(data.describe())
```

Step 3: Apply Hierarchical Clustering

```
from sklearn.preprocessing import StandardScaler
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.cluster import AgglomerativeClustering

# Standardize the data
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data.iloc[:, :-1])

# Apply hierarchical clustering (Agglomerative clustering)
linkage_matrix = linkage(scaled_data, method='ward')

# Create a Dendrogram to visualize the hierarchical clustering
import matplotlib.pyplot as plt
dendrogram(linkage_matrix)
plt.title('Dendrogram')
plt.xlabel('Samples')
plt.ylabel('Distance')
plt.show()
```

Step 4: Choose Number of Clusters and Apply

```
# Create a hierarchical clustering model with a defined number of clusters (k=3 for the Iris dataset)
hierarchical_model = AgglomerativeClustering(n_clusters=3, linkage='ward') # Remove affinity argument
```

```
# Fit the model and assign cluster labels

data['cluster'] = hierarchical_model.fit_predict(scaled_data)


# View the first few rows with cluster labels

print(data.head())
```

Step 5: Visualize the Clustering Results

```
import seaborn as sns


# Scatter plot of Sepal length vs. Sepal width, colored by cluster

sns.scatterplot(data=data, x='sepal length (cm)', y='sepal width (cm)', hue='cluster',
palette='viridis')

plt.title('Hierarchical Clustering - Iris Dataset')

plt.show()
```

Results

The hierarchical clustering algorithm successfully grouped the data into three clusters. A dendrogram was used to visualize the clustering process. The final clusters were assigned based on the distance threshold in the linkage matrix.

Conclusion

Hierarchical clustering provides a powerful method to visualize and group data without needing a predefined number of clusters. The Dendrogram helps in determining the appropriate number of clusters by observing the distance at which data points merge.

Viva Questions

1. What is hierarchical clustering, and how does it differ from K-means clustering?
2. What is the significance of the linkage method in hierarchical clustering?
3. How do you interpret a dendrogram, and how can it help in determining the number of clusters?
4. What are the advantages and limitations of hierarchical clustering?

6. Implement Density-Based Clustering Algorithm (DBSCAN) on a Dataset

Objective

To apply the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm to a dataset to identify clusters and outliers based on density.

Prerequisites

- Python programming.
- Libraries: pandas, numpy, sklearn, matplotlib, seaborn.

Dataset Description

- **Dataset Used:** Iris dataset.
- **Source:** Available in the sklearn library.
- **Features:**
    - Sepal length
    - Sepal width
    - Petal length
    - Petal width
- **Target:** Species of Iris flowers (Setosa, Versicolor, Virginica).

Procedure

Step 1: Load the Dataset

```
from sklearn.datasets import load_iris

import pandas as pd


# Load dataset

iris = load_iris()


# Create a DataFrame

data = pd.DataFrame(iris.data, columns=iris.feature_names)

data['target'] = iris.target


# View the first few rows

print(data.head())
```

Step 2: Explore the Dataset

```
# Basic information
print(data.info())

# Summary statistics
print(data.describe())
```

Step 3: Apply DBSCAN for Density-Based Clustering

```
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import DBSCAN

# Standardize the data
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data.iloc[:, :-1])

# Apply DBSCAN
dbscan = DBSCAN(eps=0.5, min_samples=5)
data['cluster'] = dbscan.fit_predict(scaled_data)

# View the first few rows with cluster labels
print(data.head())
```

- eps (epsilon): Defines the maximum distance between two samples for them to be considered as in the same neighborhood.
- min_samples: The number of samples in a neighborhood for a point to be considered as a core point.

Step 4: Analyze the Results

```
# Number of clusters and noise points
n_clusters = len(set(data['cluster'])) - (1 if -1 in data['cluster'] else 0)
n_noise = list(data['cluster']).count(-1)
```

```
print(f"Number of clusters: {n_clusters}")
print(f"Number of noise points: {n_noise}")
```

Points labeled as -1 are considered noise or outliers by DBSCAN.

Step 5: Visualize the Clustering Results

```
import seaborn as sns
import matplotlib.pyplot as plt


# Scatter plot of Sepal length vs. Sepal width, colored by cluster
sns.scatterplot(data=data, x='sepal length (cm)', y='sepal width (cm)', hue='cluster',
palette='viridis')
plt.title('DBSCAN Clustering - Iris Dataset')
plt.show()
```

Results

The DBSCAN algorithm successfully grouped the data into clusters based on density. Outliers or noise points are labeled as -1. The clustering results were visualized using a scatter plot of Sepal length vs. Sepal width.

Conclusion

DBSCAN is a powerful density-based clustering algorithm that identifies clusters of varying shapes and sizes and detects outliers as noise points. Unlike K-means, DBSCAN does not require specifying the number of clusters in advance. The eps and min_samples parameters control the density and the formation of clusters.

Viva Questions

1. What is DBSCAN, and how does it differ from K-means clustering?
2. What do the parameters eps and min_samples mean in DBSCAN, and how do they affect clustering results?
3. How does DBSCAN handle noise or outliers in the dataset?
4. What are the advantages and limitations of DBSCAN?

7. Implement Apriori Algorithm on a Dataset

Objective

To apply the Apriori algorithm for discovering frequent itemsets and generating association rules from a dataset.

Prerequisites

- Python programming.
- Libraries: mlxtend, pandas.

Dataset Description

- **Dataset Used:** A sample transactional dataset.
- **Source:** You can create a custom dataset or use any available market-basket dataset.
- **Features:**
    - Each transaction is represented as a list of items bought by a customer.

Procedure

Step 1: Install Required Libraries

To use the Apriori algorithm, we need the mlxtend library. If you don't have it installed, you can install it via pip:

```
pip install mlxtend
```

Step 2: Load the Dataset

```
import pandas as pd


# Sample transactional data (each row represents a transaction with items bought)

data = [

    ['Milk', 'Bread', 'Butter'],

    ['Beer', 'Diaper', 'Milk', 'Bread'],

    ['Milk', 'Diaper', 'Beer', 'Cola'],

    ['Bread', 'Milk', 'Butter'],

    ['Diaper', 'Milk', 'Bread', 'Butter']

]


# Convert to a DataFrame
```

```
df = pd.DataFrame(data, columns=['Item 1', 'Item 2', 'Item 3', 'Item 4'])
df = df.apply(lambda x: x.dropna().tolist(), axis=1)

# View the first few transactions
print(df.head())
```

Step 3: Preprocess the Data for Apriori

We need to convert the dataset into a format that can be processed by the Apriori algorithm. This is usually done by converting the dataset into a one-hot encoded format.

```
from mlxtend.preprocessing import TransactionEncoder

# Convert list of transactions into one-hot encoded format
te = TransactionEncoder()
te_ary = te.fit(df).transform(df)

# Convert to a DataFrame
encoded_df = pd.DataFrame(te_ary, columns=te.columns_)
print(encoded_df.head())
```

Step 4: Apply the Apriori Algorithm

Now, we apply the Apriori algorithm to find frequent itemsets with a specified minimum support.

```
from mlxtend.frequent_patterns import apriori

# Apply apriori algorithm to find frequent itemsets with a minimum support of 0.6
frequent_itemsets = apriori(encoded_df, min_support=0.6, use_colnames=True)

# View the frequent itemsets
print(frequent_itemsets)
```

Step 5: Generate Association Rules

Once the frequent itemsets are found, we generate association rules based on the frequent itemsets.

```
from mlxtend.frequent_patterns import association_rules


# Generate association rules with a minimum lift of 1.2

rules = association_rules(frequent_itemsets, metric="lift", min_threshold=1.2)


# View the generated rules

print(rules)
```

Step 6: Visualize the Results (Optional)

You can visualize the association rules using a plot.

```
import matplotlib.pyplot as plt

import seaborn as sns


# Plot a heatmap of the association rules' lift

sns.heatmap(rules.pivot_table(index='antecedents', columns='consequents', values='lift'), annot=True)

plt.title("Heatmap of Association Rules' Lift")

plt.show()
```

Results

The Apriori algorithm identifies frequent itemsets based on the minimum support threshold, and the association rules show relationships between items that often appear together in transactions. The lift metric indicates the strength of the rule.

Conclusion

The Apriori algorithm is widely used in market basket analysis and other fields where association rule mining is necessary. It helps in finding patterns in large datasets, such as items that are frequently bought together. The minimum support and lift thresholds are key parameters that control the output.

Viva Questions

1. What is the Apriori algorithm, and how does it work?
2. What is the meaning of support, confidence, and lift in the context of association rules?
3. How does the Apriori algorithm handle large datasets?
4. What are the advantages and limitations of the Apriori algorithm?

8. Implement FP-Growth Algorithm on a Dataset

Objective

To apply the FP-Growth (Frequent Pattern Growth) algorithm to find frequent itemsets in a dataset. The FP-Growth algorithm is an efficient alternative to the Apriori algorithm, particularly useful for large datasets.

Prerequisites

- Python programming.
- Libraries: mlxtend, pandas.

Dataset Description

- **Dataset Used:** A sample transactional dataset (market-basket data).
- **Source:** You can create a custom dataset or use any available dataset.
- **Features:**
    - Each transaction is represented as a list of items bought by a customer.

Procedure

Step 1: Install Required Libraries

To use the FP-Growth algorithm, we need the mlxtend library. If you don't have it installed, you can install it via pip:

```
pip install mlxtend
```

Step 2: Load the Dataset

```
import pandas as pd


# Sample transactional data (each row represents a transaction with items bought)

data = [

    ['Milk', 'Bread', 'Butter'],

    ['Beer', 'Diaper', 'Milk', 'Bread'],

    ['Milk', 'Diaper', 'Beer', 'Cola'],

    ['Bread', 'Milk', 'Butter'],

    ['Diaper', 'Milk', 'Bread', 'Butter']

]
```

```
# Convert to a DataFrame
df = pd.DataFrame(data, columns=['Item 1', 'Item 2', 'Item 3', 'Item 4'])
df = df.apply(lambda x: x.dropna().tolist(), axis=1)

# View the first few transactions
print(df.head())
```

Step 3: Preprocess the Data for FP-Growth

We need to convert the dataset into a format that can be processed by the FP-Growth algorithm. This is usually done by converting the dataset into a one-hot encoded format.

```
from mlxtend.preprocessing import TransactionEncoder

# Convert list of transactions into one-hot encoded format
te = TransactionEncoder()
te_ary = te.fit(df).transform(df)

# Convert to a DataFrame
encoded_df = pd.DataFrame(te_ary, columns=te.columns_)
print(encoded_df.head())
```

Step 4: Apply the FP-Growth Algorithm

Now, we apply the FP-Growth algorithm to find frequent itemsets with a specified minimum support.

```
from mlxtend.frequent_patterns import fpgrowth

# Apply FP-Growth algorithm to find frequent itemsets with a minimum support of 0.6
frequent_itemsets = fpgrowth(encoded_df, min_support=0.6, use_colnames=True)

# View the frequent itemsets
```

```
print(frequent_itemsets)
```

Step 5: Generate Association Rules (Optional)

Once the frequent itemsets are found, you can generate association rules based on these itemsets.

```
from mlxtend.frequent_patterns import association_rules

# Generate association rules with a minimum lift of 1.2
# Added 'num_itemsets' argument based on your version of mlxtend
rules = association_rules(frequent_itemsets, metric="lift", min_threshold=1.2, num_itemsets=2)

# View the generated rules
print("\nAssociation Rules:")
print(rules)
```

Step 6: Visualize the Results (Optional)

You can visualize the association rules using a plot.

```
import matplotlib.pyplot as plt

import seaborn as sns


# Plot a heatmap of the association rules' lift

sns.heatmap(rules.pivot_table(index='antecedents', columns='consequents', values='lift'),
annot=True)

plt.title("Heatmap of Association Rules' Lift")

plt.show()
```

Results

The FP-Growth algorithm identifies frequent itemsets based on the minimum support threshold. It is more efficient than the Apriori algorithm and does not generate candidate itemsets, making it faster for larger datasets. The association rules, if generated, show relationships between items that often appear together in transactions.

Conclusion

The FP-Growth algorithm is highly efficient for mining frequent itemsets in large datasets. It reduces the computational complexity compared to the Apriori algorithm by eliminating the need to generate candidate itemsets. This algorithm is ideal for large-scale market-basket analysis and other applications requiring frequent pattern mining.

Viva Questions

1. What is the FP-Growth algorithm, and how does it differ from the Apriori algorithm?
2. What is the concept of support in frequent itemset mining?
3. How does the FP-Growth algorithm handle the problem of candidate generation in Apriori?
4. How do the parameters support and lift affect the results in FP-Growth?

9. Implement Decision Tree Classification Algorithm on a Labeled Dataset

Objective

To implement the Decision Tree classification algorithm to classify data based on a labeled dataset.

Prerequisites

- Python programming.
- Libraries: pandas, numpy, sklearn, matplotlib, seaborn.

Dataset Description

- **Dataset Used:** A sample labeled dataset (e.g., the Iris dataset).
- **Source:** The dataset can be loaded directly from sklearn.datasets.
- **Features:**
    - The Iris dataset contains four features: sepal length, sepal width, petal length, and petal width.
    - Target: The species of the Iris plant (Setosa, Versicolor, or Virginica).

Procedure

Step 1: Install Required Libraries

Make sure you have the required libraries installed. You can install them using pip:

```
pip install pandas numpy scikit-learn matplotlib seaborn
```

Step 2: Load the Dataset

We will use the famous Iris dataset available in sklearn.

```
import pandas as pd

from sklearn.datasets import load_iris


# Load the Iris dataset

iris = load_iris()


# Convert to a DataFrame

data = pd.DataFrame(iris.data, columns=iris.feature_names)

data['target'] = iris.target
```

```
# View the first few rows
print(data.head())
```

Step 3: Split the Dataset into Training and Testing Sets

We will split the dataset into training and testing sets to evaluate the performance of the model.

```
from sklearn.model_selection import train_test_split

# Split the data into features and target
X = data.drop('target', axis=1)
y = data['target']

# Split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print("Training set size:", X_train.shape)
print("Test set size:", X_test.shape)
```

Step 4: Train the Decision Tree Model

Now, we will initialize and train the Decision Tree classifier using the training data.

```
from sklearn.tree import DecisionTreeClassifier

# Initialize the Decision Tree Classifier
dt = DecisionTreeClassifier(random_state=42)

# Train the classifier
dt.fit(X_train, y_train)
```

Step 5: Make Predictions

After training the model, we can use it to predict the target values on the test data.

```
# Predict on the test set
y_pred = dt.predict(X_test)
```

```
# Print the predictions
print("Predicted labels:", y_pred)
```

Step 6: Evaluate the Model

We will evaluate the model using accuracy, confusion matrix, and classification report.

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", conf_matrix)

# Classification Report
class_report = classification_report(y_test, y_pred)
print("Classification Report:\n", class_report)
```

Step 7: Visualize the Decision Tree (Optional)

It’s useful to visualize the decision tree to understand how it makes decisions.

```
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

# Plot the Decision Tree
plt.figure(figsize=(12, 8))
plot_tree(dt, filled=True, feature_names=iris.feature_names, class_names=iris.target_names,
rounded=True, fontsize=12)
```

```
plt.show()
```

Results

- **Accuracy:** The percentage of correctly predicted labels on the test set.
- **Confusion Matrix:** A table showing the correct and incorrect predictions broken down by class.
- **Classification Report:** A report containing metrics like precision, recall, and F1-score for each class.

Conclusion

The Decision Tree classifier is an effective model for classification tasks, particularly for small datasets. It’s interpretable, as shown in the visualized decision tree, which can help in understanding the decisions the model makes.

Viva Questions

1. What is a Decision Tree, and how does it work?
2. How do you evaluate a classification model, and what metrics are important?
3. What are the advantages and disadvantages of using Decision Trees for classification?
4. How can you prevent overfitting in Decision Trees?

10. Implement k-Nearest Neighbors (k-NN) Classification Algorithm on a Labeled Dataset

Objective

To implement the k-Nearest Neighbors (k-NN) classification algorithm on a labeled dataset and evaluate its performance.

Prerequisites

- Python programming.
- Libraries: pandas, numpy, sklearn, matplotlib, seaborn.

Dataset Description

- **Dataset Used:** Iris dataset (commonly used for classification tasks).
- **Source:** The dataset is available in the sklearn.datasets module.
- **Features:**
    - The Iris dataset consists of four features: sepal length, sepal width, petal length, and petal width.
    - **Target:** Species of the Iris plant (Setosa, Versicolor, or Virginica).

Procedure

Step 1: Install Required Libraries

Ensure you have the necessary libraries installed using pip:

```
pip install pandas numpy scikit-learn matplotlib seaborn
```

Step 2: Load the Dataset

We'll use the Iris dataset, which is available in sklearn.

```
import pandas as pd

from sklearn.datasets import load_iris


# Load the Iris dataset

iris = load_iris()


# Convert to a DataFrame

data = pd.DataFrame(iris.data, columns=iris.feature_names)

data['target'] = iris.target
```

```
# View the first few rows
print(data.head())
```

Step 3: Split the Dataset into Training and Testing Sets

We will split the data into training and testing sets (80% for training, 20% for testing) for model evaluation.

```
from sklearn.model_selection import train_test_split


# Split the data into features (X) and target (y)
X = data.drop('target', axis=1)
y = data['target']


# Split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


print("Training set size:", X_train.shape)
print("Test set size:", X_test.shape)
```

Step 4: Train the k-NN Model

Now, we'll initialize and train the k-NN classifier using the training data.

```
from sklearn.neighbors import KNeighborsClassifier


# Initialize the k-NN Classifier with k=3
knn = KNeighborsClassifier(n_neighbors=3)


# Train the classifier
knn.fit(X_train, y_train)
```

Step 5: Make Predictions

After training the model, use it to predict the class labels on the test set.

```
# Predict on the test set
y_pred = knn.predict(X_test)

# Print the predictions
print("Predicted labels:", y_pred)
```

Step 6: Evaluate the Model

We will evaluate the model's performance using accuracy, confusion matrix, and classification report.

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", conf_matrix)

# Classification Report
class_report = classification_report(y_test, y_pred)
print("Classification Report:\n", class_report)
```

Step 7: Visualize the k-NN Classifier (Optional)

It can be helpful to visualize how the classifier is performing. Here's how to visualize the decision boundaries for a 2D feature subset (e.g., petal length vs. petal width).

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

# Take a subset of the data (only 2 features for visualization)
X_vis = X.iloc[:, [2, 3]]  # Using petal length and petal width
X_train_vis, X_test_vis, y_train_vis, y_test_vis = train_test_split(X_vis, y, test_size=0.2,
random_state=42)

# Train the k-NN classifier on the subset of features
knn_vis = KNeighborsClassifier(n_neighbors=3)
knn_vis.fit(X_train_vis, y_train_vis)

# Create a mesh grid for plotting decision boundaries
h = .02  # Step size in the mesh
x_min, x_max = X_vis.iloc[:, 0].min() - 1, X_vis.iloc[:, 0].max() + 1
y_min, y_max = X_vis.iloc[:, 1].min() - 1, X_vis.iloc[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

# Plot decision boundaries
Z = knn_vis.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot the decision boundary and scatter points
plt.contourf(xx, yy, Z, alpha=0.8)
plt.scatter(X_vis.iloc[:, 0], X_vis.iloc[:, 1], c=y, edgecolors='k', marker='o', s=100)
plt.title("k-NN Classifier Decision Boundaries")
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.show()
```

Results

- **Accuracy:** The proportion of correctly classified samples in the test set.
- **Confusion Matrix:** A matrix showing how many instances from each class are classified correctly or incorrectly.
- **Classification Report:** Includes precision, recall, and F1-score for each class.

Conclusion

The k-NN classifier is a simple, yet powerful algorithm for classification. By selecting an appropriate value for k, it can perform well on various datasets. Visualizing the decision boundaries helps understand how the classifier separates the classes based on feature values.

Viva Questions

1. What is the k-NN algorithm and how does it work?
2. What is the role of the parameter k in k-NN, and how does it affect model performance?
3. How do you choose the best value of k for the k-NN classifier?
4. What are the advantages and disadvantages of using k-NN for classification?

11. Implement Support Vector Machines (SVM) Classification Algorithm on a Labeled Dataset

Objective

To implement the Support Vector Machines (SVM) classification algorithm on a labeled dataset and evaluate its performance.

Prerequisites

- Python programming.
- Libraries: pandas, numpy, sklearn, matplotlib, seaborn.

Dataset Description

- **Dataset Used:** Iris dataset (commonly used for classification tasks).
- **Source:** The dataset is available in the sklearn.datasets module.
- **Features:**
  - The Iris dataset consists of four features: sepal length, sepal width, petal length, and petal width.
  - **Target:** Species of the Iris plant (Setosa, Versicolor, or Virginica).

Procedure

Step 1: Install Required Libraries

Ensure you have the necessary libraries installed using pip:

```
pip install pandas numpy scikit-learn matplotlib seaborn
```

Step 2: Load the Dataset

We will use the Iris dataset, which is available in sklearn.

```
import pandas as pd

from sklearn.datasets import load_iris


# Load the Iris dataset

iris = load_iris()


# Convert to a DataFrame

data = pd.DataFrame(iris.data, columns=iris.feature_names)

data['target'] = iris.target
```

```
# View the first few rows
print(data.head())
```

Step 3: Split the Dataset into Training and Testing Sets

We will split the data into training and testing sets (80% for training, 20% for testing) for model evaluation.

```
from sklearn.model_selection import train_test_split


# Split the data into features (X) and target (y)
X = data.drop('target', axis=1)
y = data['target']


# Split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


print("Training set size:", X_train.shape)
print("Test set size:", X_test.shape)
```

Step 4: Train the SVM Model

Now, we'll initialize and train the SVM classifier using the training data.

```
from sklearn.svm import SVC


# Initialize the Support Vector Classifier (SVC)
svm = SVC(kernel='linear')  # You can also try 'poly', 'rbf', etc. for different kernels


# Train the classifier
svm.fit(X_train, y_train)
```

Step 5: Make Predictions

After training the model, use it to predict the class labels on the test set.

```
# Predict on the test set

y_pred = svm.predict(X_test)


# Print the predictions

print("Predicted labels:", y_pred)
```

Step 6: Evaluate the Model

We will evaluate the model's performance using accuracy, confusion matrix, and classification report.

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report


# Accuracy of the model

accuracy = accuracy_score(y_test, y_pred)

print("Accuracy:", accuracy)


# Confusion Matrix

conf_matrix = confusion_matrix(y_test, y_pred)

print("Confusion Matrix:\n", conf_matrix)


# Classification Report

class_report = classification_report(y_test, y_pred)

print("Classification Report:\n", class_report)
```

Step 7: Visualize the SVM Classifier (Optional)

To better understand how the classifier performs, you can visualize the decision boundaries for a 2D feature subset (e.g., petal length vs. petal width).

```
import numpy as np

import matplotlib.pyplot as plt


# Take a subset of the data (only 2 features for visualization)
```

```
X_vis = X.iloc[:, [2, 3]]  # Using petal length and petal width

X_train_vis, X_test_vis, y_train_vis, y_test_vis = train_test_split(X_vis, y, test_size=0.2,
random_state=42)


# Train the SVM classifier on the subset of features

svm_vis = SVC(kernel='linear')

svm_vis.fit(X_train_vis, y_train_vis)


# Create a mesh grid for plotting decision boundaries

h = .02  # Step size in the mesh

x_min, x_max = X_vis.iloc[:, 0].min() - 1, X_vis.iloc[:, 0].max() + 1

y_min, y_max = X_vis.iloc[:, 1].min() - 1, X_vis.iloc[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x_min, x_max, h),

                     np.arange(y_min, y_max, h))


# Plot decision boundaries

Z = svm_vis.predict(np.c_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)


# Plot the decision boundary and scatter points

plt.contourf(xx, yy, Z, alpha=0.8)

plt.scatter(X_vis.iloc[:, 0], X_vis.iloc[:, 1], c=y, edgecolors='k', marker='o', s=100)

plt.title("SVM Classifier Decision Boundaries")

plt.xlabel('Petal Length')

plt.ylabel('Petal Width')

plt.show()
```

Results

- **Accuracy:** The proportion of correctly classified samples in the test set.
- **Confusion Matrix:** A matrix showing how many instances from each class are classified correctly or incorrectly.

- **Classification Report:** Includes precision, recall, and F1-score for each class.

Conclusion

The SVM classifier is effective for classification tasks, particularly when there is a clear margin of separation between classes. The linear kernel works well in simple cases, but other kernels like 'rbf' or 'poly' can be tried for more complex data.

Viva Questions

1. What is the SVM algorithm and how does it work?
2. What is the role of the kernel in SVM, and how do you choose it?
3. What is the difference between a linear kernel and a radial basis function (RBF) kernel in SVM?
4. How do you tune the parameters C and gamma in SVM?

12. Implement Naïve Bayes Classification Algorithm on a Labeled Dataset

Objective

To implement the Naïve Bayes classification algorithm on a labeled dataset and evaluate its performance.

Prerequisites

- Python programming.
- Libraries: pandas, numpy, sklearn, matplotlib, seaborn.

Dataset Description

- **Dataset Used:** Iris dataset (commonly used for classification tasks).
- **Source:** The dataset is available in the sklearn.datasets module.
- **Features:**
    - The Iris dataset consists of four features: sepal length, sepal width, petal length, and petal width.
    - **Target:** Species of the Iris plant (Setosa, Versicolor, or Virginica).

Procedure

Step 1: Install Required Libraries

Ensure you have the necessary libraries installed using pip:

```
pip install pandas numpy scikit-learn matplotlib seaborn
```

Step 2: Load the Dataset

We will use the Iris dataset, which is available in sklearn.

```
import pandas as pd

from sklearn.datasets import load_iris


# Load the Iris dataset

iris = load_iris()


# Convert to a DataFrame

data = pd.DataFrame(iris.data, columns=iris.feature_names)

data['target'] = iris.target
```

```
# View the first few rows
print(data.head())
```

Step 3: Split the Dataset into Training and Testing Sets

We will split the data into training and testing sets (80% for training, 20% for testing) for model evaluation.

```
from sklearn.model_selection import train_test_split


# Split the data into features (X) and target (y)
X = data.drop('target', axis=1)
y = data['target']


# Split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


print("Training set size:", X_train.shape)
print("Test set size:", X_test.shape)
```

Step 4: Train the Naïve Bayes Model

Now, we'll initialize and train the Naïve Bayes classifier using the training data.

```
from sklearn.naive_bayes import GaussianNB


# Initialize the Gaussian Naïve Bayes classifier
nb = GaussianNB()


# Train the classifier
nb.fit(X_train, y_train)
```

Step 5: Make Predictions

After training the model, use it to predict the class labels on the test set.

```
# Predict on the test set
```

```
y_pred = nb.predict(X_test)


# Print the predictions
print("Predicted labels:", y_pred)
```

Step 6: Evaluate the Model

We will evaluate the model's performance using accuracy, confusion matrix, and classification report.

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report


# Accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)


# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", conf_matrix)


# Classification Report
class_report = classification_report(y_test, y_pred)
print("Classification Report:\n", class_report)
```

Step 7: Visualize the Results (Optional)

We can visualize the confusion matrix using a heatmap.

```
import seaborn as sns
import matplotlib.pyplot as plt


# Plot confusion matrix using seaborn heatmap
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=iris.target_names,
yticklabels=iris.target_names)
```

```
plt.title('Confusion Matrix')

plt.xlabel('Predicted Label')

plt.ylabel('True Label')

plt.show()
```

Results

- **Accuracy:** The proportion of correctly classified samples in the test set.
- **Confusion Matrix:** A matrix showing how many instances from each class are classified correctly or incorrectly.
- **Classification Report:** Includes precision, recall, and F1-score for each class.

Conclusion

The Naïve Bayes classifier is effective for classification tasks, especially when the features are conditionally independent. It's a simple yet powerful model for tasks like text classification, medical diagnosis, and more.

Viva Questions

1. What is the Naïve Bayes algorithm and how does it work?
2. What is the assumption made by the Naïve Bayes classifier regarding features?
3. What is the difference between Gaussian Naïve Bayes and Multinomial Naïve Bayes?
4. How do you handle categorical and continuous data in Naïve Bayes?

13. Build and Evaluate a Linear Regression Model to Predict a Continuous Target Variable

Objective

To implement a Linear Regression model to predict a continuous target variable and evaluate its performance using metrics such as Mean Squared Error (MSE) and R-squared.

Prerequisites

- Python programming.
- Libraries: pandas, numpy, sklearn, matplotlib, seaborn.

Dataset Description

- **Dataset Used:** Boston Housing dataset (commonly used for regression tasks).
- **Source:** The dataset is available in the sklearn.datasets module.
- **Features:**
  - Various features of housing data, such as crime rate, average number of rooms, property tax rate, etc.
  - **Target:** Median value of owner-occupied homes (in thousands of dollars).

Procedure

Step 1: Install Required Libraries

Ensure you have the necessary libraries installed using pip:

```
pip install pandas numpy scikit-learn matplotlib seaborn
```

Step 2: Load the Dataset

We'll use the Boston Housing dataset.

```
import pandas as pd

from sklearn.datasets import load_boston


# Load the Boston Housing dataset

boston = load_boston()


# Convert to a DataFrame

data = pd.DataFrame(boston.data, columns=boston.feature_names)

data['target'] = boston.target
```

```
# View the first few rows
print(data.head())
```

Step 3: Split the Dataset into Training and Testing Sets

We will split the data into training and testing sets (80% for training, 20% for testing) for model evaluation.

```
from sklearn.model_selection import train_test_split


# Split the dataset into features (X) and target (y)
X = data.drop('target', axis=1)
y = data['target']


# Split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


print("Training set size:", X_train.shape)
print("Test set size:", X_test.shape)
```

Step 4: Train the Linear Regression Model

Now, we'll initialize and train the linear regression model using the training data.

```
from sklearn.linear_model import LinearRegression


# Initialize the Linear Regression model
model = LinearRegression()


# Train the model
model.fit(X_train, y_train)
```

Step 5: Make Predictions

After training the model, we use it to predict the target variable on the test set.

```
# Predict on the test set
```

```
y_pred = model.predict(X_test)


# Print the predictions
print("Predicted values:", y_pred[:5])
```

Step 6: Evaluate the Model

We will evaluate the model using performance metrics like Mean Squared Error (MSE), R-squared, and the residual plot.

```
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt


# Mean Squared Error (MSE)
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)


# R-squared value
r2 = r2_score(y_test, y_pred)
print("R-squared:", r2)


# Residual plot (for visual evaluation)
residuals = y_test - y_pred
plt.figure(figsize=(8, 6))
plt.scatter(y_pred, residuals)
plt.axhline(y=0, color='r', linestyle='--')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residual Plot')
plt.show()
```

Step 7: Visualize the Results (Optional)

Visualizing the predictions vs actual values can provide more insight into the model's performance.

```
# Plotting Actual vs Predicted values

plt.figure(figsize=(8, 6))

plt.scatter(y_test, y_pred)

plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='red', linestyle='--')

plt.xlabel('Actual Values')

plt.ylabel('Predicted Values')

plt.title('Actual vs Predicted Values')

plt.show()
```

Results

- **Mean Squared Error (MSE):** The average of the squared differences between the predicted and actual values. Lower values indicate better performance.
- **R-squared:** A measure of how well the model explains the variance in the data. Values closer to 1 indicate a better fit.
- **Residual Plot:** A visualization of the residuals (errors). It helps detect patterns that might indicate issues with the model.

Conclusion

The Linear Regression model has been successfully trained and evaluated on the Boston Housing dataset. The evaluation metrics, such as R-squared and MSE, help assess the model's accuracy. Visualizations like residual and actual-vs-predicted plots can offer deeper insights into the model's performance.

Viva Questions

1. What is Linear Regression, and how does it work?
2. What is the significance of R-squared in regression analysis?
3. What is the difference between simple linear regression and multiple linear regression?
4. How can you improve a linear regression model if the performance is not satisfactory?

14. Build and Evaluate a Polynomial Regression Model

Objective

To implement a **Polynomial Regression** model to predict a continuous target variable and evaluate its performance using metrics such as Mean Squared Error (MSE) and R-squared.

Prerequisites

- Python programming.
- Libraries: pandas, numpy, sklearn, matplotlib.

Dataset Description

We will use the **Boston Housing dataset**, which is often used for regression tasks.

- **Features:** Various features of housing data, such as crime rate, average number of rooms, property tax rate, etc.
- **Target:** Median value of owner-occupied homes (in thousands of dollars).

Procedure

Step 1: Install Required Libraries

Ensure you have the necessary libraries installed using pip:

```
pip install pandas numpy scikit-learn matplotlib
```

Step 2: Load the Dataset

We will use the **Boston Housing dataset**.

```
import pandas as pd

from sklearn.datasets import load_boston


# Load the Boston Housing dataset

boston = load_boston()


# Convert to a DataFrame

data = pd.DataFrame(boston.data, columns=boston.feature_names)

data['target'] = boston.target
```

```
# View the first few rows
print(data.head())
```

Step 3: Split the Dataset into Training and Testing Sets

We will split the data into training and testing sets (80% for training, 20% for testing) for model evaluation.

```
from sklearn.model_selection import train_test_split


# Split the dataset into features (X) and target (y)
X = data.drop('target', axis=1)
y = data['target']


# Split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


print("Training set size:", X_train.shape)
print("Test set size:", X_test.shape)
```

Step 4: Polynomial Feature Transformation

To apply polynomial regression, we need to transform the features into higher-degree polynomial terms.

```
from sklearn.preprocessing import PolynomialFeatures


# Initialize the PolynomialFeatures transformer with degree 2
poly = PolynomialFeatures(degree=2)


# Fit and transform the training data to create polynomial features
X_train_poly = poly.fit_transform(X_train)
```

```
# Transform the test data
X_test_poly = poly.transform(X_test)

# View the transformed features
print("Transformed training features:\n", X_train_poly[:5])
```

Step 5: Train the Polynomial Regression Model

Now, we will fit the Polynomial Regression model to the transformed features.

```
from sklearn.linear_model import LinearRegression

# Initialize the Linear Regression model
model = LinearRegression()

# Train the model on the polynomial features
model.fit(X_train_poly, y_train)
```

Step 6: Make Predictions

Now that the model is trained, we use it to predict the target variable on the test set.

```
# Predict on the test set using the polynomial features
y_pred = model.predict(X_test_poly)

# Print the predictions
print("Predicted values:", y_pred[:5])
```

Step 7: Evaluate the Model

We will evaluate the model using performance metrics like Mean Squared Error (MSE), R-squared, and visualize the results.

```
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt

# Mean Squared Error (MSE)
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)

# R-squared value
r2 = r2_score(y_test, y_pred)
print("R-squared:", r2)

# Residual plot (for visual evaluation)
residuals = y_test - y_pred
plt.figure(figsize=(8, 6))
plt.scatter(y_pred, residuals)
plt.axhline(y=0, color='r', linestyle='--')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residual Plot for Polynomial Regression')
plt.show()
```

Step 8: Visualize the Results (Optional)

We can also visualize how well the model fits the actual vs predicted values.

```
# Plotting Actual vs Predicted values
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred)
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='red', linestyle='--')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Actual vs Predicted Values for Polynomial Regression')

plt.show()
```

Results

- **Mean Squared Error (MSE):** This is a measure of how close the predictions are to the actual values. A lower value indicates better model performance.
- **R-squared:** This value indicates how well the polynomial regression model explains the variance in the data. Values closer to 1 are better.
- **Residual Plot:** The plot of residuals helps assess how well the model's predictions match the actual values.

Conclusion

The Polynomial Regression model has been successfully trained and evaluated on the Boston Housing dataset. The **evaluation metrics** such as **MSE** and **R-squared** help assess the model's performance, while the **residual plot** gives insight into the accuracy of the predictions.

Viva Questions

1. What is polynomial regression, and how does it differ from linear regression?
2. How do polynomial features help improve the performance of a regression model?
3. What is the impact of the degree of polynomial features on the model's performance?
4. Why is it important to evaluate the model using metrics like MSE and R-squared?