INDEX

EXPERIMENT – 1

AIM : To install Anaconda Distribution and become familiar with essential data science libraries for Python experimentation.

Apparatus Required:

1. Computer with internet connectivity
2. Anaconda Distribution installer (downloaded from https://www.anaconda.com/products/distribution)
3. Python IDE (Integrated Development Environment) such as Jupyter Notebook or JupyterLab

Theory:

Anaconda Distribution is a free and open-source distribution of Python and R programming languages for data science and machine learning tasks. It includes various libraries and tools pre-installed, making it easy for data scientists and developers to get started with their projects without worrying about dependency management.

Procedure:

1. Download the Anaconda Distribution installer from the official website (https://www.anaconda.com/products/distribution) based on your operating system (Windows, macOS, or Linux).
2. Run the installer and follow the on-screen instructions to install Anaconda Distribution on your system
3. Once the installation is complete, open a terminal (Command Prompt on Windows) and create a new conda environment using the command:

   ```
   conda create --name myenv
   ```

   Replace **myenv** with the desired name for your environment.
4. Activate the newly created environment using the command:

   ```
   conda activate myenv
   ```

5. Install essential data science libraries such as NumPy, Pandas, Matplotlib, and Scikit-learn using the following commands:

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

6. Once the installation is complete, launch a Python IDE such as Jupyter Notebook or JupyterLab by typing the command:

   ```
   jupyter notebook
   ```

   This will open a new browser window/tab with the Jupyter Notebook interface.

7. Create a new notebook and start experimenting with the installed libraries. You can import the libraries using the following Python code:

   ```
   import numpy as np
   import pandas as pd
   import matplotlib.pyplot as plt
   import sklearn
   ```

Observation and Results:

Upon successful completion of the installation and experimentation, you should be able to:

- Create and manipulate arrays and matrices using NumPy.
- Perform data analysis and manipulation tasks using Pandas.
- Create various types of visualizations using Matplotlib.
- Apply machine learning algorithms for classification, regression, and clustering using Scikit-learn.

EXPERIMENT – 2

AIM : The aim of this experiment is to study and understand the usage of Python libraries commonly used for machine learning applications, including Pandas, NumPy, and Matplotlib.

Software Used:

1. Python (Programming Language)
2. Jupyter Notebook (Integrated Development Environment)
3. Anaconda Distribution (Python Distribution for Data Science)

Theory:

1. **Pandas:**
   Pandas is a powerful data manipulation and analysis library for Python. It provides data structures like Series and DataFrame, which are widely used for data exploration, cleaning, and transformation tasks. Pandas simplifies the process of working with structured data and offers functionalities for reading and writing data from various file formats such as CSV, Excel, SQL databases, and more.

2. **NumPy:**
   NumPy is the fundamental package for scientific computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. NumPy's array object, ndarray, allows for fast numerical computations and is the cornerstone of many other Python libraries used in data science and machine learning.

3. **Matplotlib:**
   Matplotlib is a comprehensive library for creating static, interactive, and animated visualizations in Python. It offers a wide range of plotting functions and customization options to create publication-quality figures for data analysis and presentation. Matplotlib can be used to generate various types of plots, including line plots, scatter plots, bar charts, histograms, and more.

Procedure:

1. **Pandas:**
   - Learn the basics of Pandas, including Series, DataFrame, and Index objects.
   - Practice data manipulation operations such as filtering, selecting, sorting, and grouping data.
   - Explore data visualization capabilities using Pandas built-in functions.
   -
2. **NumPy:**
   - Understand the fundamentals of NumPy arrays and its advantages over native Python lists.
   - Learn about basic array operations, including arithmetic, slicing, and reshaping.
   - Explore advanced features such as broadcasting and universal functions (ufuncs).

3. **Matplotlib:**
   - Familiarize yourself with the Matplotlib library for data visualization.
   - Learn to create various types of plots, including line plots, scatter plots, histograms, and bar charts.
   - Experiment with customizing plot styles, colors, labels, and annotations.

EXPERIMENT – 3

AIM : To perform data preprocessing like outlier detection, handling missing values, data normalization, and class balance.

Software Used:

1. Python (Programming Language)
2. Jupyter Notebook (Integrated Development Environment)
3. Pandas (Python Library for Data Manipulation)
4. NumPy (Python Library for Numerical Computing)
5. Scikit-learn (Python Library for Machine Learning)

Theory:

Data preprocessing is a crucial step in the machine learning pipeline. It involves transforming raw data into a format that is suitable for machine learning algorithms. The following techniques will be covered in this experiment:

1. Outlier Detection: Identifying and handling outliers in the dataset to prevent them from affecting the model's performance.
2. Handling Missing Values: Strategies for dealing with missing data, such as imputation or removal of incomplete samples.
3. Data Normalization: Scaling numerical features to a standard range to ensure that all features contribute equally to the model.
4. Class Balance: Techniques for addressing class imbalance in classification tasks, such as oversampling, undersampling, or using class weights.

Procedure:

1. **Load Dataset:** Load the raw dataset containing features and target variables.
2. **Outlier Detection:** Identify outliers in the dataset using statistical methods or visualization techniques.
3. **Handling Missing Values:** Fill in missing values using imputation techniques such as mean, median, or mode imputation.

4. **Data Normalization:** Scale numerical features to have a mean of 0 and standard deviation of 1 using StandardScaler.
5. **Class Balance:** Address class imbalance issues by applying appropriate techniques based on the dataset characteristics.

Code:

```python
# Python code for data preprocessing

import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer

# Load dataset
data = pd.read_csv('dataset.csv')

# Outlier Detection
# Implement outlier detection techniques such as z-score, IQR, or visualization-based methods
# Example using z-score:
z_scores = (data - data.mean()) / data.std()
outliers = z_scores[abs(z_scores) > 3]

# Handling Missing Values
imputer = SimpleImputer(strategy='mean')
data_filled = pd.DataFrame(imputer.fit_transform(data), columns=data.columns)

# Data Normalization
scaler = StandardScaler()
data_normalized = pd.DataFrame(scaler.fit_transform(data_filled), columns=data_filled.columns)

# Class Balance
# Implement techniques to address class imbalance, such as oversampling, undersampling, or using class weights
# Example using class weights:
from sklearn.utils.class_weight import compute_class_weight
class_weights = compute_class_weight( class_weight: 'balanced', data['target'].unique(), data['target'])

```

Code Explanation:

1. **Import required libraries:**
   - **import pandas as pd:** Imports the pandas library for data manipulation and analysis.
   - **from sklearn.preprocessing import StandardScaler:** Imports the StandardScaler class from the scikit-learn library for data preprocessing, specifically for scaling features.
   - **from sklearn.impute import SimpleImputer:** Imports the SimpleImputer class from scikit-learn for handling missing values.

2. **Load dataset:**
   - **data = pd.read_csv('dataset.csv'):** Reads the dataset from a CSV file named 'dataset.csv' and loads it into a pandas DataFrame named 'data'.

3. **Outlier Detection:**
   - This section implements outlier detection techniques. However, the specific techniques are not provided in the code. Common methods include z-score, IQR, or visualization-based methods.
   - The code example suggests the use of z-score for outlier detection, where z-scores are calculated for each data point and outliers are identified based on a threshold of 3 standard deviations.

4. **Handling Missing Values:**
   - **imputer = SimpleImputer(strategy='mean'):** Creates an instance of SimpleImputer with the strategy set to 'mean', which replaces missing values with the mean of each column.
   - **data_filled = pd.DataFrame(imputer.fit_transform(data), columns=data.columns):** Uses the SimpleImputer instance to fill missing values in the dataset and stores the result in a new DataFrame named 'data_filled'.

5. **Data Normalization:**
   - **scaler = StandardScaler():** Creates an instance of StandardScaler for data normalization.
   - **data_normalized = pd.DataFrame(scaler.fit_transform(data_filled), columns=data_filled.columns):** Normalizes the filled dataset using z-scores and stores the normalized data in a new DataFrame named 'data_normalized'.

6. **Class Balance:**
   - This section addresses class imbalance in the dataset, but the specific technique is not provided in the code.
   - The example suggests the use of class weights to handle class imbalance, where the **compute_class_weight** function from scikit-learn is used to compute the class weights. However, the target variable ('target') is assumed to be present in the dataset, and its unique values are used for computing class weights.

EXPERIMENT – 4

AIM : To perform the k-Nearest Neighbor and Simple Linear Regression algorithm to perform data classification and analyse the results using confusion matrix accuracy, sensitivity, and specificity parameters.

Software Used:

1. Python programming language
2. Scikit-learn library for implementing k-NN and Simple Linear Regression
3. Matplotlib library for visualization
4. Numpy and Pandas for data manipulation and analysis

Theory:

k-Nearest Neighbor (k-NN) Algorithm:

1. k-NN is a non-parametric and instance-based learning algorithm for classification and regression tasks.
2. It works by finding the k-nearest neighbors of a given data point and making predictions based on the majority class (classification) or average (regression) of its neighbors.
3. Key considerations include choosing an appropriate value for k, understanding its impact on model performance, and being aware of the algorithm's sensitivity to the choice of distance metric.

Simple Linear Regression:

1. Simple Linear Regression is a linear regression model that assumes a linear relationship between the independent and dependent variables.
2. It estimates the relationship using a straight line that minimizes the sum of squared differences between observed and predicted values.
3. Model interpretation involves understanding the slope (weight) and intercept parameters, assessing model fit using metrics like R-squared, and validating assumptions such as linearity and homoscedasticity.

Procedure:

1. Load the dataset and preprocess if necessary (handling missing values, encoding categorical variables).
2. Split the dataset into training and testing sets.
3. Implement k-NN algorithm using the KNeighborsClassifier class from Scikit-learn, specifying the number of neighbors (k).
4. Train the Simple Linear Regression model using the LinearRegression class from Scikit-learn.
5. Evaluate the models using confusion matrix, accuracy, sensitivity, and specificity parameters.
6. Visualize the results using Matplotlib for better understanding.

Code:

```python
# Import necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LinearRegression
from sklearn.metrics import confusion_matrix, accuracy_score

# Load dataset - Replace 'dataset.csv' with the appropriate file path
data = pd.read_csv('exp4.csv')

# Preprocess the data if necessary

# Split data into features (X) and target (y)
X = data.drop(columns=['Purchased'])  # Features excluding the 'Purchased' column
y = data['Purchased']  # Target variable

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split( *arrays: X, y, test_size=0.2, random_state=42)

# Implement k-Nearest Neighbor algorithm
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# Implement Simple Linear Regression
linear_reg = LinearRegression()
linear_reg.fit(X_train, y_train)

# Evaluate the models
# Confusion matrix for k-NN
knn_pred = knn.predict(X_test)
knn_conf_matrix = confusion_matrix(y_test, knn_pred)
```

```
knn_conf_matrix = confusion_matrix(y_test, knn_pred)

# Accuracy for k-NN
knn_accuracy = accuracy_score(y_test, knn_pred)

# Print confusion matrix and accuracy for k-NN
print("Confusion Matrix for k-Nearest Neighbor:")
print(knn_conf_matrix)
print("Accuracy for k-Nearest Neighbor:", knn_accuracy)

# Simple Linear Regression predictions
linear_reg_pred = linear_reg.predict(X_test)

# Output sample results for Simple Linear Regression
print("Sample output for Simple Linear Regression predictions:")
print(linear_reg_pred)

```

Code Explanation:

1. **Importing Libraries:**
   - The code starts by importing necessary libraries such as pandas for data manipulation and scikit-learn modules for machine learning tasks.

2. **Loading Dataset:**
   - The dataset is loaded from a CSV file named 'dataset.csv' using the pd.read_csv() function from pandas.

3. **Splitting Data:**
   - The data is split into features (X) and the target variable (y) using the drop() function to exclude the target column from features and indexing to select the target column.

4. **Splitting into Training and Testing Sets:**
   - The dataset is split into training and testing sets using the train_test_split() function from scikit-learn. It splits the data into 80% training and 20% testing sets.

5. **Implementing k-Nearest Neighbor (k-NN) Algorithm:**
   - The k-NN algorithm is implemented with KNeighborsClassifier() from scikit-learn, setting the number of neighbors to 5, and trained on the training data using the fit() method.

6. **Implementing Simple Linear Regression:**
   - Simple Linear Regression is implemented using LinearRegression() from scikit-learn and trained on the training data using the fit() method.

7. **Model Evaluation:**
   - Confusion matrix and accuracy score are computed for the k-NN algorithm using the confusion_matrix() and accuracy_score() functions from scikit-learn.

8. **Printing Sample Results:**
   - Sample results are printed, including the confusion matrix for k-NN and the accuracy score. Additionally, the predictions from the Simple Linear Regression model are printed as a sample output.

Ouput:

```
Confusion Matrix for k-Nearest Neighbor:
[[14  1]
 [ 1  4]]
Accuracy for k-Nearest Neighbor: 0.9

Sample output for Simple Linear Regression predictions:
[ 0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  0.1]
```

EXPERIMENT – 5

AIM : To perform the Support Vector Machines and Naïve Bayes algorithm for data classification and analyse the results using confusion matrix accuracy, recall, precision, F1 score parameters.

Software Used:

1. Python
2. Jupyter Notebook or any Python IDE
3. Libraries: pandas, scikit-learn

Theory:

1. Support Vector Machines (SVM):

- SVM is a supervised machine learning algorithm used for classification and regression tasks.
- It works by finding the hyperplane that best separates the classes in feature space.
- SVM aims to maximize the margin between the classes, making it robust to outliers.
- SVM can handle high-dimensional data efficiently, making it suitable for tasks with a large number of features.
- It offers various kernel functions (e.g., linear, polynomial, radial basis function) to handle non-linear decision boundaries, enhancing its flexibility in capturing complex patterns in the data.

2. Naïve Bayes Algorithm:

- Naïve Bayes is a probabilistic classification algorithm based on Bayes' theorem.
- It assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.
- Despite its simplicity and strong independence assumptions, Naïve Bayes often performs well in practice, especially for text classification tasks.
- Naïve Bayes is computationally efficient and can be trained quickly even with large datasets, making it suitable for real-time applications.

Procedure:

1. **Data Loading and Preprocessing:**
   - Load the dataset into a DataFrame.
   - Perform any necessary preprocessing steps.

2. **Model Training and Evaluation:**
   - Split the data into training and testing sets.
   - Train SVM and Naïve Bayes classifiers on the training data.
   - Evaluate the performance of both classifiers using various metrics.

3. **Results Analysis:**
   - Analyze the confusion matrices and performance metrics to understand the classification results.
   - Compare the performance of SVM and Naïve Bayes algorithms.

Code:

```
# Importing necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix, accuracy_score, recall_score, precision_score, f1_score

# Reading the dataset
data = pd.read_csv('exp5.csv')

# Splitting the data into features (X) and target (y)
X = data.drop(columns=['Purchased'])
y = data['Purchased']

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split( *arrays: X, y, test_size=0.2, random_state=42)

# Implementing Support Vector Machines (SVM) algorithm
svm_model = SVC(kernel='linear')
svm_model.fit(X_train, y_train)

# Implementing Naïve Bayes algorithm
nb_model = GaussianNB()
nb_model.fit(X_train, y_train)

# Evaluating the models
# Confusion matrix and other metrics for SVM
svm_pred = svm_model.predict(X_test)
svm_conf_matrix = confusion_matrix(y_test, svm_pred)
svm_accuracy = accuracy_score(y_test, svm_pred)
```

```
svm_accuracy = accuracy_score(y_test, svm_pred)
svm_recall = recall_score(y_test, svm_pred)
svm_precision = precision_score(y_test, svm_pred)
svm_f1 = f1_score(y_test, svm_pred)

# Confusion matrix and other metrics for Naïve Bayes
nb_pred = nb_model.predict(X_test)
nb_conf_matrix = confusion_matrix(y_test, nb_pred)
nb_accuracy = accuracy_score(y_test, nb_pred)
nb_recall = recall_score(y_test, nb_pred)
nb_precision = precision_score(y_test, nb_pred)
nb_f1 = f1_score(y_test, nb_pred)

# Outputting sample results for SVM
print("SVM Results:")
print("Confusion Matrix:")
print(svm_conf_matrix)
print("Accuracy:", svm_accuracy)
print("Recall:", svm_recall)
print("Precision:", svm_precision)
print("F1 Score:", svm_f1)

# Outputting sample results for Naïve Bayes
print("\nNaïve Bayes Results:")
print("Confusion Matrix:")
print(nb_conf_matrix)
print("Accuracy:", nb_accuracy)
print("Recall:", nb_recall)
print("Precision:", nb_precision)
print("F1 Score:", nb_f1)
```

Code Explanation:

1. **Importing Libraries:** It imports necessary libraries such as pandas for data manipulation, and scikit-learn functions for model training and evaluation.

2. **Loading Dataset:** It reads the dataset from a CSV file named 'exp5.csv' using pandas and stores it in a DataFrame called data.

3. **Data Preprocessing:** It converts the categorical variable 'Gender' into numerical values. 'Male' is mapped to 0 and 'Female' is mapped to 1. This is done to convert categorical data into a format suitable for machine learning algorithms.

4. **Splitting Data:** It splits the dataset into features (X) and the target variable (y), and then further splits them into training and testing sets using the train_test_split function from scikit-learn. The testing set size is set to 20% of the total dataset, and a random state is specified for reproducibility.

5. **Model Training:** It trains two classification models - Support Vector Machines (SVM) and Naïve Bayes - using the training data.

6. **Model Evaluation:** It evaluates the performance of the trained models using various metrics such as accuracy, recall, precision, and F1 score. The confusion matrix is also computed to analyze the true positives, true negatives, false positives, and false negatives.

7. **Output Results:** It prints the evaluation results for both SVM and Naïve Bayes models, including the confusion matrix and various performance metrics.

Ouput:

```
SVM Results:
Confusion Matrix:
[[16  0]
 [ 1  3]]
Accuracy: 0.95
Recall: 0.75
Precision: 1.0
F1 Score: 0.8571428571428571

Naïve Bayes Results:
Confusion Matrix:
[[15  1]
 [ 0  4]]
Accuracy: 0.95
Recall: 1.0
Precision: 0.8
F1 Score: 0.888888888888889
```

EXPERIMENT – 6

AIM : To perform data dimensionality reduction using the principal component analysis technique.

Software Used:

1. Python
   - pandas
   - scikit-learn (sklearn)
   - matplotlib (for visualization, optional)

Theory:

Principal Component Analysis (PCA) is a widely used technique in the field of data analysis and machine learning. It is particularly useful when dealing with high-dimensional datasets, as it helps in reducing the number of features while retaining the essential information present in the data. PCA works by transforming the original features into a new set of orthogonal features called principal components. These components are ordered by the amount of variance they explain in the data, with the first component explaining the most variance and subsequent components explaining less.

PCA operates under the assumption that the directions (or axes) in the feature space that capture the most variance are the most informative, and therefore, the most important for describing the dataset's structure. By projecting the data onto a lower-dimensional subspace defined by these principal components, PCA effectively reduces the dimensionality of the dataset while preserving as much of the variance as possible.

Procedure:

1. **Import Libraries:** Import the necessary Python libraries for data manipulation, PCA, and visualization (if required).

2. **Load Dataset:** Load the dataset you want to perform dimensionality reduction on using pandas.

3. **Preprocess Data:** Preprocess the data if necessary, including handling missing values, encoding categorical variables, and scaling numerical features.

4. **Apply PCA:** Apply PCA to the preprocessed dataset using sklearn's PCA class. Choose the number of components based on the desired level of dimensionality reduction.

5. **Analyze Explained Variance:** Analyze the explained variance ratio of each principal component to understand how much information each component retains.

6. **Visualize Results (Optional):** Visualize the data in the reduced-dimensional space using matplotlib or any other visualization library to gain insights into the data structure.

Code:

```python
# Import necessary libraries
from sklearn import datasets  # to retrieve the iris Dataset
import pandas as pd  # to load the dataframe
from sklearn.preprocessing import StandardScaler  # to standardize the features
from sklearn.decomposition import PCA  # to apply PCA
import seaborn as sns  # to plot the heat maps

# Step 1: Import necessary libraries

# Step 2: Load the dataset
iris = datasets.load_iris()
df = pd.DataFrame(iris['data'], columns=iris['feature_names'])
df.head()

# Step 3: Standardize the features
scalar = StandardScaler()
scaled_data = pd.DataFrame(scalar.fit_transform(df))
scaled_data

# Step 4: Check the Co-relation between features without PCA (Optional)
sns.heatmap(scaled_data.corr())
# Optional: Visualization of the correlation heatmap

# Step 5: Applying Principal Component Analysis
pca = PCA(n_components=3)
pca.fit(scaled_data)
data_pca = pca.transform(scaled_data)
data_pca = pd.DataFrame(data_pca, columns=['PC1', 'PC2', 'PC3'])
data_pca.head()

# Step 6: Checking Co-relation between features after PCA
sns.heatmap(data_pca.corr())
```

Code Explanation:

1. **Import Necessary Libraries:** The required libraries are imported. These include datasets from sklearn to retrieve the iris dataset, pandas to work with dataframes, StandardScaler from sklearn.preprocessing to standardize the features, PCA from sklearn.decomposition to apply PCA, and seaborn to plot heatmaps.

2. **Load the Dataset:** The iris dataset is loaded using datasets.load_iris() method from sklearn. Then, it is converted into a pandas DataFrame df for further processing.

3. **Standardize the Features:** The features in the dataset are standardized using StandardScaler(). This ensures that all features have a mean of 0 and a standard deviation of 1, which is a prerequisite for PCA.

4. **Check the Correlation Between Features Without PCA (Optional):** A heatmap of the correlation matrix of the standardized dataset is plotted using seaborn.heatmap(). This helps visualize the relationships between features before applying PCA.

5. **Applying Principal Component Analysis (PCA):** PCA is applied to the standardized dataset using PCA(n_components=3), indicating that we want to retain three principal components. The fit() method is called to fit the PCA model to the data, and then transform() is used to obtain the reduced dataset. The resulting principal components are stored in a new DataFrame data_pca.

6. **Checking Correlation Between Features After PCA:** A heatmap of the correlation matrix of the principal components (data_pca) is plotted using seaborn.heatmap(). This helps visualize the relationships between the principal components after dimensionality reduction.

Ouputs:

1. DataFrame

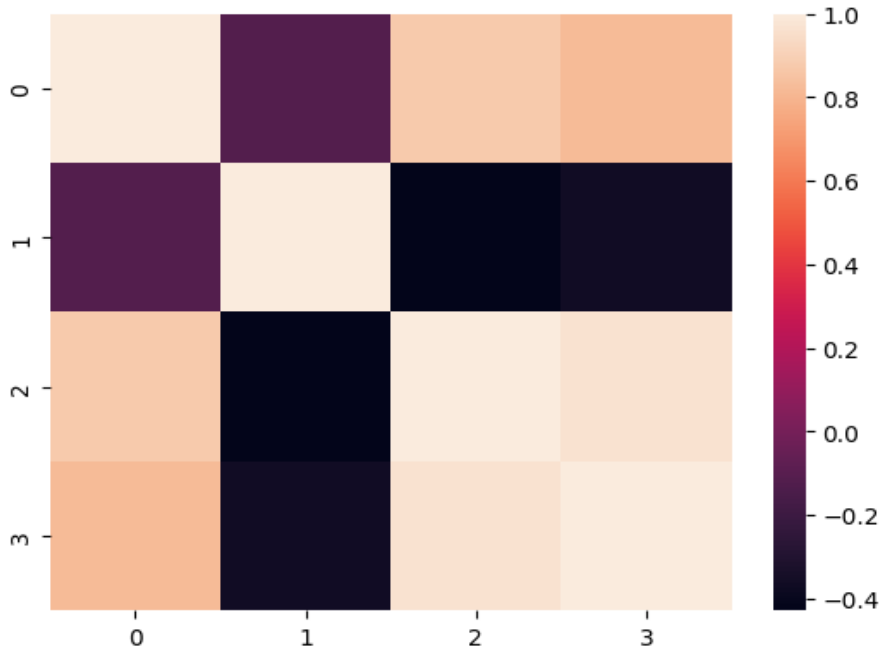
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

2. Scaled Dataset:

	0	1	2	3
0	-0.900681	1.019004	-1.340227	-1.315444
1	-1.143017	-0.131979	-1.340227	-1.315444
2	-1.385353	0.328414	-1.397064	-1.315444
3	-1.506521	0.098217	-1.283389	-1.315444
4	-1.021849	1.249201	-1.340227	-1.315444
...	...	...	...	...
145	1.038005	-0.131979	0.819596	1.448832
146	0.553333	-1.282963	0.705921	0.922303
147	0.795669	-0.131979	0.819596	1.053935
148	0.432165	0.788808	0.933271	1.448832
149	0.068662	-0.131979	0.762758	0.790671

150 rows × 4 columns

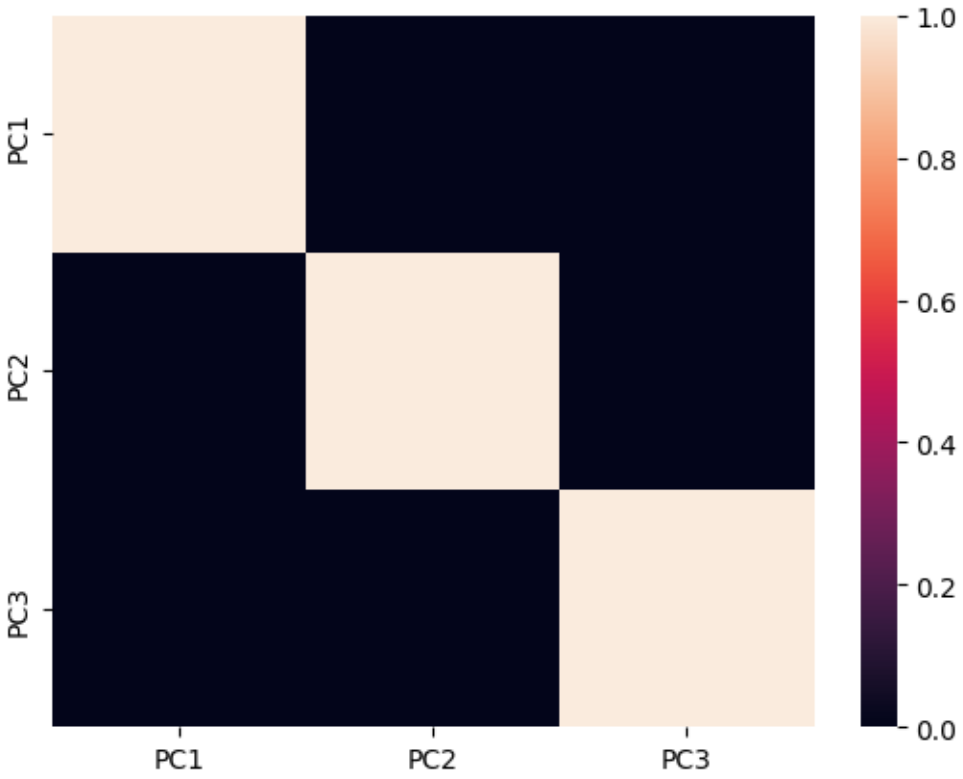
3. Correlation Heatmap of Scaled Dataset:



4. Principal Component Analysis (PCA) Results:

	PC1	PC2	PC3
0	-2.264703	0.480027	-0.127706
1	-2.080961	-0.674134	-0.234609
2	-2.364229	-0.341908	0.044201
3	-2.299384	-0.597395	0.091290
4	-2.389842	0.646835	0.015738

5. Visualization of Correlation Heatmap after PCA



EXPERIMENT – 7

AIM : To perform data classification using the ADABoost and Gradient Boosting algorithm. Use random-search hyperparameter optimization to fine-tune model's performance.

Software Used:

1. Python
2. Scikit-learn
3. Pandas
4. Seaborn

Theory:

1. **ADABoost (Adaptive Boosting):**
    - ADABoost is an ensemble learning method that fits a sequence of weak learners (usually decision trees) on repeatedly modified versions of the data.
    - It combines the outputs of weak learners to form a strong learner that makes accurate predictions.
    - ADABoost works by assigning weights to each observation in the dataset. Misclassified observations are given higher weights so that subsequent weak learners focus more on classifying those observations correctly.

2. **Gradient Boosting**
    - Gradient Boosting is another ensemble learning technique that builds a strong learner by sequentially adding new models to correct the errors made by previous models.
    - Unlike ADABoost, Gradient Boosting does not modify the weights of observations. Instead, it fits new models to the residual errors made by the previous models.
    - Gradient Boosting uses gradient descent optimization to minimize the loss function, resulting in a strong predictive model.

3. **Random Search Hyperparameter Optimization:**
    - Hyperparameters are parameters that are set before the learning process begins.
    - Random search hyperparameter optimization is a technique used to search for the best combination of hyperparameters by randomly selecting combinations from a predefined grid of hyperparameters.
    - It helps in finding optimal hyperparameters more efficiently compared to exhaustive search methods.

Procedure:

1. **Load the dataset:** Import the dataset containing the features and target variable.

2. **Preprocess the data:** Handle any missing values, encode categorical variables if present, and perform any other necessary preprocessing steps to prepare the data for modeling.

3. **Split the dataset:** Divide the dataset into features (X) and target (y) variables, where X contains the independent variables and y contains the dependent variable.

4. **Split the data into training and testing sets:** Use the train_test_split function from scikit-learn to split the dataset into training and testing sets. This allows us to train the model on a portion of the data and evaluate its performance on unseen data.

5. **Initialize the classifiers:** Create instances of the AdaBoostClassifier and GradientBoostingClassifier from scikit-learn.

6. **Define the hyperparameter grid:** Specify the hyperparameters and their corresponding values that we want to tune during the random search optimization. This is typically done using a dictionary where keys are hyperparameters and values are lists of possible values for each hyperparameter.

7. **Perform random search hyperparameter optimization:** Use scikit-learn's RandomizedSearchCV to search for the best combination of hyperparameters for both classifiers. Random search randomly samples from the hyperparameter grid and evaluates the models' performance using cross-validation.

8. **Fit the optimized models:** Fit the optimized AdaBoost and Gradient Boosting models on the training data using the best hyperparameters obtained from the random search.

9. **Evaluate the models:** Assess the performance of the models on the testing data using appropriate evaluation metrics such as accuracy, precision, recall, and F1-score. This helps us understand how well the models generalize to unseen data.

10. **Compare the performance:** Compare the performance of the AdaBoost and Gradient Boosting classifiers based on the evaluation metrics. This allows us to determine which algorithm performs better for the given dataset.

Codes and Outputs:

1. Hyperparameter Tuning using Grid Seach CV

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
# Import necessary libraries
from sklearn.model_selection import GridSearchCV

# Load the Titanic dataset
titanic_data = pd.read_csv("exp7.csv")

# Let's do some basic preprocessing for simplicity
# Replace missing values and encode categorical variables
titanic_data.fillna( value: 0, inplace=True)
titanic_data = pd.get_dummies(titanic_data, columns=['Sex', 'Embarked'], drop_first=True)

# Select features and target variable
X = titanic_data.drop( labels: ['Survived', 'PassengerId', 'Name', 'Ticket', 'Cabin'], axis=1)
y = titanic_data['Survived']

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

# Define the parameter grid for GridSearchCV
param_grid = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7],
}

# Initialize the Gradient Boosting model
gb_model = GradientBoostingClassifier()
```

```
# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=gb_model, param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1)

# Fit the model to the training data using GridSearchCV
grid_search.fit(X_train, y_train)

# Get the best parameters and best model
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_

# Make predictions on the test set using the best model
y_pred_best = best_model.predict(X_test)

# Evaluate the best model
accuracy_best = accuracy_score(y_test, y_pred_best)

# Print the results
print("Best Parameters:", best_params)
print(f"Best Model Accuracy: {accuracy_best}")
```

Output:

```
Best Parameters: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 200}
Best Model Accuracy: 0.8044692737430168

Process finished with exit code 0
```

2. Hyperparameter Tuning using Randomized Search CV

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
# Import necessary libraries
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
import numpy as np

# Load the Titanic dataset
titanic_data = pd.read_csv("exp7.csv")

# Let's do some basic preprocessing for simplicity
# Replace missing values and encode categorical variables
titanic_data.fillna( value: 0, inplace=True)
titanic_data = pd.get_dummies(titanic_data, columns=['Sex', 'Embarked'], drop_first=True)

# Select features and target variable
X = titanic_data.drop( labels: ['Survived', 'PassengerId', 'Name', 'Ticket', 'Cabin'], axis=1)
y = titanic_data['Survived']

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

# Define the parameter grid for RandomizedSearchCV
param_dist = {
    'n_estimators': np.arange(50, 251, 50),
    'learning_rate': np.linspace( start: 0.01,  stop: 0.2,  num: 10),
    'max_depth': np.arange(3, 8),
}
```

```
# Initialize the Gradient Boosting model
gb_model = GradientBoostingClassifier()

# Initialize RandomizedSearchCV
random_search = RandomizedSearchCV(estimator=gb_model, param_distributions=param_dist, n_iter=10,
                                   cv=5, scoring='accuracy', random_state=42, n_jobs=-1)

# Fit the model to the training data using RandomizedSearchCV
random_search.fit(X_train, y_train)

# Get the best parameters and best model
best_params_random = random_search.best_params_
best_model_random = random_search.best_estimator_

# Make predictions on the test set using the best model
y_pred_best_random = best_model_random.predict(X_test)

# Evaluate the best model
accuracy_best_random = accuracy_score(y_test, y_pred_best_random)

# Print the results
print("Best Parameters (Randomized Search):", best_params_random)
print(f"Best Model Accuracy (Randomized Search): {accuracy_best_random}")
```

Output:

```
Best Parameters (Randomized Search): {'n_estimators': 250, 'max_depth': 3, 'learning_rate': 0.09444444444444444}
Best Model Accuracy (Randomized Search): 0.8156424581005587

Process finished with exit code 0
```

3. AdABoost Classifier

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
# Import necessary libraries
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
import numpy as np

# Load the Titanic dataset
titanic_data = pd.read_csv("exp7.csv")

# Let's do some basic preprocessing for simplicity
# Replace missing values and encode categorical variables
titanic_data.fillna( value: 0, inplace=True)
titanic_data = pd.get_dummies(titanic_data, columns=['Sex', 'Embarked'], drop_first=True)

# Select features and target variable
X = titanic_data.drop( labels: ['Survived', 'PassengerId', 'Name', 'Ticket', 'Cabin'], axis=1)
y = titanic_data['Survived']

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

# Define the parameter grid for RandomizedSearchCV
param_dist = {
    'n_estimators': np.arange(50, 251, 50),
    'learning_rate': np.linspace( start: 0.01,  stop: 0.2,  num: 10),
    'algorithm': ['SAMME', 'SAMME.R']
}
```

```python

# Initialize the AdaBoost model
ada_model = AdaBoostClassifier()

# Initialize RandomizedSearchCV
random_search = RandomizedSearchCV(estimator=ada_model, param_distributions=param_dist, n_iter=10,
                                   cv=5, scoring='accuracy', random_state=42, n_jobs=-1)

# Fit the model to the training data using RandomizedSearchCV
random_search.fit(X_train, y_train)

# Get the best parameters and best model
best_params_random = random_search.best_params_
best_model_random = random_search.best_estimator_

# Make predictions on the test set using the best model
y_pred_best_random = best_model_random.predict(X_test)

# Evaluate the best model
accuracy_best_random = accuracy_score(y_test, y_pred_best_random)

# Print the results
print("Best Parameters (Randomized Search):", best_params_random)
print(f"Best Model Accuracy (Randomized Search): {accuracy_best_random}")

```

Output:

```
Best Parameters (Randomized Search): {'n_estimators': 50, 'learning_rate': 0.0944444444444444, 'algorithm': 'SAMME.R'}
Best Model Accuracy (Randomized Search): 0.7821229050279329

Process finished with exit code 0
```

EXPERIMENT – 8

AIM : To perform data classification using the Decision tree algorithm and analyse the model's performance using AUC-ROC curve. Use grid-search hyperparameter optimization to fine-tune model's performance.

Software Used:

1. Python

Theory:

Decision trees are powerful machine learning algorithms used for both classification and regression tasks. They work by partitioning the feature space into regions, where each region corresponds to a leaf node in the tree, and predictions are made based on the majority class or average target value of the instances within that region.

The AUC-ROC curve (Area Under the Receiver Operating Characteristic curve) is a performance metric used for binary classification problems. It represents the trade-off between the true positive rate (sensitivity) and the false positive rate (1 - specificity) across different threshold values. A higher AUC-ROC score indicates better classifier performance.

Procedure:

1. **Data Preparation:** Clean and preprocess the dataset, handling missing values and encoding categorical variables.
2. **Splitting the Dataset:** Divide the dataset into a training set and a testing set for model evaluation.
3. **Decision Tree Construction:** Build the decision tree model by recursively partitioning the feature space based on a splitting criterion.
4. **Model Training:** Train the decision tree model on the training data to learn the decision rules.
5. **Model Evaluation:** Evaluate the trained model's performance on the testing set using evaluation metrics like accuracy, precision, recall, F1-score, and ROC-AUC score.
6. **Hyperparameter Tuning:** Fine-tune the model's hyperparameters using techniques like grid search or randomized search to optimize its performance.

7. **Model Interpretation:** Interpret the decision tree model to understand its decision-making process and visualize the learned decision rules.
8. **Analysis and Iteration:** Analyze the model's performance and iteratively refine it by adjusting hyperparameters, preprocessing techniques, or feature selection methods.

EXPERIMENT – 9

AIM : To implement Random Forest classification and check if the model is well-fitting. Study the conditions for over-fit and under-fit for a model.

Software Used:

1. Python programming language
2. Scikit-learn library for machine learning tasks

Theory:

Random Forest is an ensemble learning method that leverages the collective wisdom of multiple decision trees to make predictions. Unlike individual decision trees, which are prone to overfitting, Random Forests create a forest of trees, each trained on a random subset of the data and features. This randomness helps in promoting diversity among the trees and reduces the risk of overfitting. During training, each tree is built by recursively splitting the data based on feature thresholds to maximize information gain (for classification) or minimize impurity (for regression). Once trained, the Random Forest aggregates the predictions of all the individual trees to make the final prediction. For classification tasks, it selects the mode of the classes predicted by the trees, while for regression tasks, it calculates the mean prediction. This ensemble approach tends to yield more robust and accurate predictions compared to individual decision trees, making Random Forest a popular choice for various machine learning tasks.

Conditions for Overfitting and Underfitting:

- **Overfitting:** Overfitting occurs when the model learns the training data too well, capturing noise and irrelevant patterns that don't generalize well to new, unseen data.
- **Underfitting:** Underfitting occurs when the model is too simple to capture the underlying structure of the data, resulting in poor performance on both the training and test data.

Procedure:

1. **Prepare the dataset:** Load the dataset and preprocess it if necessary (e.g., handling missing values, encoding categorical variables).

2. **Split the dataset:** Divide the dataset into training and testing sets.

3. **Train the Random Forest model:** Fit the Random Forest classifier to the training data.

4. **Evaluate the model:** Predict the classes on the test data and assess the model's performance using metrics such as accuracy, precision, recall, and F1-score.

5. **Check for overfitting and underfitting:** Analyze the model's performance on the training and test data. Overfitting is indicated by high accuracy on the training data but lower accuracy on the test data, while underfitting is indicated by poor performance on both datasets.

Code:

```python
# Importing necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report

# Load dataset (replace 'dataset.csv' with the actual filename)
data = pd.read_csv('exp9.csv')

# Drop irrelevant columns like PassengerId, Name, Ticket, and Cabin
data.drop( labels: ['PassengerId', 'Name', 'Ticket', 'Cabin'], axis=1, inplace=True)

# Convert categorical variables (Sex, Embarked) into numerical representations
data['Sex'] = data['Sex'].map({'male': 0, 'female': 1})
data['Embarked'] = data['Embarked'].map({'S': 0, 'C': 1, 'Q': 2})

# Handle missing values
data.fillna(method='ffill', inplace=True)  # Forward fill missing values

# Split dataset into features and target variable
X = data.drop( labels: 'Survived', axis=1)  # Features
y = data['Survived']  # Target variable

# Split dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split( *arrays: X, y, test_size=0.2, random_state=42)

# Initialize Random Forest classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the Random Forest classifier
rf_classifier.fit(X_train, y_train)
```

```
32
33  # Predict on the test data
34  y_pred = rf_classifier.predict(X_test)
35
36  # Evaluate model performance
37  accuracy = accuracy_score(y_test, y_pred)
38  print("Accuracy:", accuracy)
39
40  # Classification report
41  print("Classification Report:")
42  print(classification_report(y_test, y_pred))
43
```

Code Explaination:

1. **Importing Necessary Libraries:**
   - It imports the required libraries for data manipulation, model evaluation, and machine learning algorithms. These include pandas for data handling, sklearn's train_test_split for splitting the dataset, RandomForestClassifier for building the Random Forest model, and accuracy_score and classification_report for evaluating the model's performance.

2. **Load Dataset:**
   - It loads the dataset from a CSV file named 'exp9.csv' into a pandas DataFrame named 'data'.

3. **Data Preprocessing:**
   - It removes irrelevant columns such as 'PassengerId', 'Name', 'Ticket', and 'Cabin' from the dataset since they are not likely to be predictive of survival.
   - It converts categorical variables 'Sex' and 'Embarked' into numerical representations. For example, it maps 'male' to 0 and 'female' to 1 for the 'Sex' column, and 'S' to 0, 'C' to 1, and 'Q' to 2 for the 'Embarked' column.

4. **Handling Missing Values:**
   - It handles missing values in the dataset by forward-filling them. This means that missing values are replaced with the previous non-missing value in the same column.

5. **Split Dataset:**
   - It splits the dataset into features (X) and the target variable (y). Here, 'X' contains all columns except for the 'Survived' column, which is the target variable stored in 'y'.
   - It further splits the dataset into training and testing sets using the train_test_split function. The testing set size is set to 20% of the entire dataset, and a random state of 42 is used for reproducibility.

6. **Initialize and Train Random Forest Classifier:**
   - It initializes a Random Forest classifier with 100 decision trees and a random state of 42 for reproducibility.
   - It trains the Random Forest classifier on the training data (X_train, y_train) using the fit method.

7. **Model Evaluation:**
   - It uses the trained Random Forest classifier to make predictions on the test data (X_test) using the predict method.
   - It calculates the accuracy of the model by comparing the predicted labels (y_pred) with the true labels (y_test) using the accuracy_score function.
   - It generates a classification report using the classification_report function, which provides metrics such as precision, recall, F1-score, and support for each class.

Output:

```
Accuracy: 0.8571428571428571
Classification Report:
              precision    recall  f1-score   support

           0       0.50      1.00      0.67         1
           1       1.00      0.83      0.91         6

    accuracy                           0.86         7
   macro avg       0.75      0.92      0.79         7
weighted avg       0.93      0.86      0.87         7


Process finished with exit code 0
```

EXPERIMENT – 10

AIM : To examine the time-series data using time-domain features extraction techniques (mean, median, mode, standard deviation, skewness, variance) and develop the classification model using the suitable ML algorithms.

Software Used:

1. Python programming language
2. Scikit-learn library for machine learning tasks

Theory:

Time-series data consists of observations collected over time. Analyzing time-series data involves extracting various statistical features from the data to understand its patterns and behavior. Some commonly used time-domain features include:

- **Mean:** The average value of the data points.
- **Median:** The middle value of the data when arranged in ascending order.
- **Mode:** The most frequently occurring value in the data.
- **Standard deviation:** A measure of the dispersion of data points around the mean.
- **Skewness:** A measure of the asymmetry of the distribution of data points.
- **Variance:** The average of the squared differences from the mean.

These features provide insights into the central tendency, spread, and shape of the time-series data, which can be useful for classification tasks.

Procedure:

1. **Load Time-Series Data:** Import the time-series dataset from a CSV file into a pandas DataFrame.

2. **Extract Time-Domain Features:** Compute statistical features such as mean, median, mode, standard deviation, skewness, and variance for each time-series instance.

3. **Prepare Target Variable:** Identify and store the target variable (e.g., class labels) separately from the dataset.

4. **Split Dataset:** Divide the dataset into training and testing sets using the train_test_split function, specifying the test size and random state.

5. **Initialize Classification Model:** Choose a suitable classification algorithm (e.g., Random Forest) and initialize the classifier with appropriate parameters.

6. **Train the Model:** Train the classification model on the training data using the fit method of the classifier.

7. **Make Predictions:** Utilize the trained model to predict the classes of the test data using the predict method.

8. **Evaluate Model Performance:** Assess the performance of the classification model using evaluation metrics like accuracy score and a classification report.

9. **Interpret Results:** Analyze the accuracy and other performance metrics to understand the effectiveness of the model.

Code:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from scipy.stats import skew  # Import the skew function

# Load time-series dataset
data = pd.read_csv('exp11.csv')

# Prepare target variable
target_column = 'Feature_1'  # Replace 'Feature_1' with the actual target column name
y = data[target_column]

# Extract time-domain features
features = pd.DataFrame()
features['mean'] = data.mean(axis=1)  # Calculate mean along rows (axis=1)
features['median'] = data.median(axis=1)
features['mode'] = data.mode(axis=1).iloc[:, 0]  # Select the first mode from the DataFrame
features['std_dev'] = data.std(axis=1)
features['skewness'] = data.apply(skew, axis=1)  # Calculate skewness along rows
features['variance'] = data.var(axis=1)

# Check if the number of samples in features and target matches
if len(features) != len(y):
    raise ValueError("Number of samples in features and target variable doesn't match.")

```

```
53
54  # Split dataset into training and testing sets
55  X_train, X_test, y_train, y_test = train_test_split( *arrays: features, y, test_size=0.2, random_state=42)
56
57  # Initialize Random Forest regressor
58  rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)
59
60  # Train the Random Forest regressor
61  rf_regressor.fit(X_train, y_train)
62
63  # Predict on the test data
64  y_pred = rf_regressor.predict(X_test)
65
66  # Evaluate model performance
67  mse = mean_squared_error(y_test, y_pred)
68  print("Mean Squared Error:", mse)
69
70  # You can add more evaluation metrics if needed
71
72
```

Code Explaination:

1. **Importing Libraries:**
   - **pandas:** Used for data manipulation and analysis.
   - **train_test_split:** From scikit-learn, used to split the dataset into training and testing sets.
   - **RandomForestRegressor:** A machine learning model from scikit-learn, used for regression tasks.
   - **mean_squared_error:** A metric from scikit-learn, used to evaluate the performance of regression models.
   - **skew:** A function from SciPy, used to calculate the skewness of data.

2. **Loading Data:** The code reads a CSV file named 'exp11.csv' into a pandas DataFrame called data.

3. **Preparing Target Variable:**
   - target_column: Specifies the target variable for the regression task, which is 'Feature_1' in this case.
   - y: Extracts the target variable from the DataFrame data and stores it in y.

4. **Extracting Features:** Time-domain features are extracted from the dataset and stored in a DataFrame called features.

5. **Data Consistency Check:** It checks if the number of samples in features matches the number of samples in the target variable y. If not, it raises a ValueError.

6. **Splitting Dataset:** The dataset is split into training and testing sets using the train_test_split function. 80% of the data is used for training (X_train, y_train), and 20% for testing (X_test, y_test).

7. **Initializing and Training Random Forest Regressor**
   - A Random Forest regressor is initialized with 100 estimators and a random state of 42.
   - The model is trained on the training data (X_train, y_train) using the fit method.

8. **Making Predictions:** The trained model is used to make predictions on the test data (X_test) using the predict method, and the predictions are stored in y_pred.

9. **Evaluating Model Performance:** Mean Squared Error (MSE) is calculated between the true target values (y_test) and the predicted values (y_pred). The MSE is printed to assess the model's performance.

Output:

```
Mean Squared Error: 0.8723964663804699

Process finished with exit code 0
```