Predict the price of the Uber ride from a given pickup point to the agreed drop-off location.
Perform following tasks:
1. Pre-process the dataset.
2. Identify outliers.
3. Check the correlation.
4. Implement linear regression and random forest regression models.
5. Evaluate the models and compare their respective scores like R2, RMSE, etc.
performs the following tasks related to data analysis and machine learning using Python's popular libraries such as pandas, NumPy, Seaborn, and scikit-learn:

1. **Importing Libraries**: The code begins by importing necessary libraries for data manipulation, visualization, and machine learning. These libraries include pandas, NumPy, Seaborn, Matplotlib, and scikit-learn's modules for model selection and evaluation.

2. **Loading the Dataset**: The code loads a dataset from a CSV file named "uber.csv" using pandas' `read_csv` function and provides information about the dataset's structure, including the number of columns, data types, and the first few rows.

3. **Data Cleaning**: Data cleaning involves several steps to ensure the quality and consistency of the dataset. The following operations are performed:
   - Dropping unnecessary columns (e.g., 'Unnamed: 0', 'key').
   - Checking for missing values and removing rows with missing data using `dropna`.
   - Converting the 'pickup_datetime' column from an object data type to a datetime data type and extracting separate date and time components from it.
   - Detecting and removing incorrect coordinates (latitude and longitude) based on their valid ranges.

4. **Haversine Formula**: The code calculates the distance between the pickup and drop-off coordinates using the Haversine formula, ensuring the accuracy of the distance calculation.

5. **Outliers Handling**: The code removes outliers from the dataset. It filters trips with very large distances as well as trips with a zero distance, as they are considered outliers. It also filters cases where the fare amount is unreasonably high for a short distance and vice versa.

6. **Correlation Matrix**: A correlation matrix is generated to find the two variables that have the most interdependence (correlation) in the dataset.

7. **Standardization**: The code standardizes the independent variable ('Distance') and dependent variable ('fare_amount') to achieve more accurate results when building a linear regression model.

8. **Splitting the Dataset**: The dataset is divided into a training set and a test set using scikit-learn's `train_test_split` function.

9. **Simple Linear Regression**: A simple linear regression model is trained on the training set. The model predicts the fare amount based on the distance. The code prints performance metrics for the model, including the R-squared value, mean absolute error, mean absolute percentage error, mean squared error, and root mean squared error.

10. **Visualization**: The code includes visualizations to show the relationship between distance and fare amount using scatterplots for both the training and test datasets.

11. **Random Forest Regressor**: A Random Forest Regressor model is trained on the training set. The model is used to predict fare amounts based on distance. Similar to the simple linear regression, performance metrics are printed for this model, and a scatterplot is generated to visualize the predicted versus actual fare amounts.

12. **Tabulated Results**: The code tabulates the performance metrics of both the linear regression and Random Forest Regressor models, including their root mean squared error and R-squared values, in a DataFrame.

The code demonstrates data cleaning, outlier handling, feature engineering, model training, and performance evaluation in a data analysis and machine learning context using Python libraries.

Classify the email using the binary classification method. Email Spam detection has two states: a) Normal State – Not Spam, b) Abnormal State – Spam. Use K-Nearest Neighbors and Support Vector Machine for classification. Analyze their performance.

This code performs the following tasks related to data analysis and machine learning using Python's popular libraries such as pandas, NumPy, Seaborn, Matplotlib, scikit-learn, and scikit-learn's machine learning models:

1. **Importing Libraries**: The code begins by importing necessary libraries for data manipulation, visualization, and machine learning. These libraries include pandas, NumPy, Seaborn, Matplotlib, and scikit-learn's modules for model selection, machine learning, and metrics.

2. **Loading the Dataset**: The code loads a dataset from a CSV file named "emails.csv" using pandas' `read_csv` function and provides information about the dataset's structure, including the number of columns, data types, and the first few rows.

3. **Data Cleaning**: Data cleaning involves several steps to ensure the quality and consistency of the dataset. The following operations are performed:
   - Dropping the 'Email No.' column as it is likely to be an identifier and not useful for prediction.
   - Checking for missing values using `isna()` and reporting the count of missing values for each column.
   - Providing summary statistics about the dataset using the `describe()` method.

4. **Separating Features and Labels**: The dataset is split into two parts:
   - `X`: Independent variables (features)
   - `y`: Dependent variable (label)

5. **Splitting the Dataset**: The code further divides the dataset into a training set and a test set using scikit-learn's `train_test_split` function. The test set constitutes 15% of the entire dataset, and a random state is set for reproducibility.

6. **Machine Learning Models**: The code defines a dictionary of machine learning models to be used for classification. These models include:
   - K-Nearest Neighbors (KNN)
   - Linear Support Vector Machine (SVM)
   - Polynomial SVM
   - Radial Basis Function (RBF) SVM
   - Sigmoid SVM

   Each model is associated with its respective configuration, including hyperparameters and kernel types.

7. **Fit and Predict on Each Model**: A loop iterates through the defined machine learning models. For each model, the following steps are performed:
   - The model is trained on the training set using `fit(X_train, y_train)`.
   - Predictions are made on the test set using `predict(X_test)`.
   - The accuracy score is calculated using `metrics.accuracy_score(y_test, y_pred)` and printed for each model.

This code demonstrates the process of loading and cleaning a dataset, splitting it into training and test sets, and training multiple machine learning models to classify data. The accuracy of each model is evaluated and printed to assess its performance.

Given a bank customer, build a neural network-based classifier that can determine whether they will leave or not in the next 6 months.

This code is an example of building a binary classification model using an artificial neural network (ANN) to predict customer churn (whether customers will leave a bank or not). It uses Python libraries, including Pandas, NumPy, Seaborn, Matplotlib, Scikit-Learn, and TensorFlow (Keras).

Here's an explanation of each section of the code:

1. **Importing Libraries**: This section imports necessary libraries for data manipulation, visualization, machine learning, and deep learning. It includes pandas, NumPy, Seaborn, Matplotlib, Scikit-Learn, and TensorFlow's Keras.

2. **Loading the Dataset**: The code loads a dataset from a CSV file named "churn_modelling.csv" using Pandas. It provides information about the dataset, such as the number of columns, data types, and the first few rows.

3. **Cleaning**: Data cleaning involves several steps:
   - Dropping columns that are not relevant for modeling (e.g., 'RowNumber', 'CustomerId', 'Surname').
   - Checking for missing values using `isna()` and reporting the count of missing values for each column.
   - Providing summary statistics about the dataset using the `describe()` method.

4. **Separating Features and Labels**: The dataset is split into two parts:
   - `X`: Independent variables (features)
   - `y`: Dependent variable (label)

5. **Encoding Categorical Data**: Categorical data in columns 'Geography' and 'Gender' are encoded into numeric values using Label Encoding. The 'Geography' column is also one-hot encoded, converting string-based features into separate dimensions. The original 'Geography' column is removed.

6. **Dimensionality Reduction**: One dimension is removed from the one-hot encoding to avoid multicollinearity.

7. **Splitting the Dataset**: The dataset is split into a training set and a test set using Scikit-Learn's `train_test_split` function. The test set constitutes 20% of the entire dataset, and a random state is set for reproducibility.

8. **Normalization**: Standardization is applied to specific columns ('CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'EstimatedSalary') to ensure that all features have the same scale.

9. **Initializing & Building the Model**: A feedforward artificial neural network (ANN) model is created using Keras' Sequential model. The architecture includes several layers:
   - The input layer with ReLU activation.
   - Four hidden layers with ReLU activation.

- The output layer with Sigmoid activation for binary classification.

10. **Compiling the Model**: The model is compiled with an Adam optimizer, binary cross-entropy loss, and accuracy as the evaluation metric. The model summary is printed.

11. **Training the Model**: The ANN model is trained on the training set using `fit()`. Training includes 20 epochs and a batch size of 32.

12. **Predicting the Results**: The model is used to predict results on the test set. A threshold of 0.5 is applied to convert probabilities into binary predictions (True or False).

13. **Confusion Matrix and Accuracy**: Scikit-Learn's `confusion_matrix` and `classification_report` functions are used to evaluate the model's performance. The accuracy score is also calculated and printed.

The code demonstrates the entire process of data preprocessing, model creation, training, and evaluation for a binary classification problem using a deep learning model.

Implement Gradient Descent Algorithm to find the local minima of a function.
For example, find the local minima of the function $y=(x+3)^2$ starting from the point x=2.

This code demonstrates an implementation of gradient descent to find the local minimum of a given function. Here's an explanation of the code:

1. **Importing Libraries**: The code imports the required libraries, including `Symbol` and `lambdify` from the SymPy library, `matplotlib` for plotting, and `numpy` for numerical operations.

2. **Defining the Function**: The target function is defined as `(x+5)^2`. This is the function for which we want to find the local minimum.

3. **`gradient_descent` Function**: This function performs gradient descent to find the local minimum of a given function. It takes several parameters:
   - `function`: The function for which the minimum needs to be found.
   - `start`: The initial value for the optimization process.
   - `learn_rate`: The learning rate, which controls the step size in each iteration.
   - `n_iter`: The maximum number of iterations (default is 10,000).
   - `tolerance`: The convergence tolerance (default is 1e-06).
   - `step_size`: The initial step size (default is 1).

4. **Creating Gradient and Function**: The `function` is converted into a Python function using `lambdify`. Similarly, the gradient of the function is created as a Python function.

5. **Gradient Descent Loop**: The gradient descent process is implemented in a `while` loop. It continues until one of the following conditions is met:
   - The step size is smaller than the tolerance.
   - The number of iterations reaches the maximum limit.

   Within the loop:
   - `prev_x` stores the current value of `start`.
   - `start` is updated using the gradient descent formula: `start = start - learn_rate * gradient(prev_x)`.
   - `step_size` is calculated as the absolute change in `start` from the previous iteration.
   - The iteration count, `iters`, is incremented.
   - The new `start` value is appended to the `points` list.

6. **Printing the Local Minimum**: Once the gradient descent process is complete, the code prints the local minimum that was found.

7. **Plotting the Function and Optimization Path**:
   - A range of x-values (`x_`) is created for plotting purposes.
   - The values of the function are computed using the given range.
   - A plot is created using `matplotlib`.
   - The function curve is plotted in red.
   - The optimization path is represented by points connected by lines.

8. **Function Definition and Optimization Call**: The target function, `(x+5)^2`, is defined. Then, the `gradient_descent` function is called with specific parameters:
   - The function to minimize.
   - The initial starting point (`start`).
   - The learning rate (`learn_rate`).

- The maximum number of iterations (`n_iter`).

The code demonstrates how gradient descent can be used to find the local minimum of a function. It visually shows the optimization path and the final minimum value of the function. In this example, it finds the local minimum of the function `(x+5)^2`.

Implement K-Nearest Neighbors algorithm on diabetes.csv dataset. Compute confusion matrix, accuracy, error rate, precision and recall on the given dataset.

This code is an example of a machine learning project using the k-Nearest Neighbors (k-NN) algorithm for a diabetes classification problem. Here's an explanation of the code:

1. **Importing Libraries**: The code imports necessary libraries, including `pandas`, `numpy`, `seaborn`, `matplotlib`, and various components from the `scikit-learn` library (`train_test_split`, `StandardScaler`, `KNeighborsClassifier`, `GridSearchCV`, `confusion_matrix`, `classification_report`, and `accuracy_score`).

2. **Loading the Dataset**: It loads the dataset from a CSV file called "diabetes.csv" and inspects the dataset using the `info()` and `head()` functions to get information about the columns, data types, and initial data values.

3. **Data Cleaning**:
   - A correlation matrix is computed and visualized using `df.corr().style.background_gradient(cmap='BuGn')`. This helps identify the correlations between features.
   - Two columns, 'BloodPressure' and 'SkinThickness', are dropped from the dataset using `df.drop(['BloodPressure', 'SkinThickness'], axis=1, inplace=True)` as they may not be useful for the classification task.
   - Null values in the dataset are checked using `df.isna().sum()`.
   - A summary of the dataset statistics is provided using `df.describe()`.

4. **Data Visualization**: The code creates a histogram plot to visualize the distribution of the dataset features.

5. **Data Separation**: The dataset is separated into independent variables (X) and the dependent variable (y). `X` contains the features (all columns except the last one), and `y` contains the target variable (the last column).

6. **Splitting the Dataset**: The dataset is split into training and testing sets using `train_test_split`. The training set contains 80% of the data, and the testing set contains 20%. Random seed 8 is used for reproducibility.

7. **Standardization**: The dataset is standardized using `StandardScaler`. This ensures that all features have a mean of 0 and a standard deviation of 1. Both the training and testing sets are standardized.

8. **k-NN Model Definition and Evaluation**:
   - A `knn` function is defined to create and evaluate a k-NN model. It takes the following parameters:
     - `X_train` and `X_test`: Training and testing data.
     - `y_train` and `y_test`: Training and testing labels.
     - `neighbors`: The number of neighbors for the k-NN algorithm.
     - `power`: The power parameter used in the Minkowski distance metric.
   - The k-NN model is created using the specified number of neighbors and power.
   - The model is trained on the training data and used to make predictions on the test data.
   - Accuracy, confusion matrix, and classification report are printed to evaluate the model's performance.

9. **Hyperparameter Tuning**:

- A grid search is performed to find the best hyperparameters for the k-NN model. It searches over a range of values for the number of neighbors (`n_neighbors`) and the power parameter (`p`).
- The best estimator, best parameters, and best score are printed.

10. **Using Best Hyperparameters**: The `knn` function is called again, but this time, it uses the best hyperparameters found by the grid search for the k-NN model. It evaluates the model with these hyperparameters.

In summary, this code loads a dataset, performs data cleaning, data visualization, and standardization, and then builds a k-NN classification model. It also performs hyperparameter tuning to find the best combination of hyperparameters for the k-NN model. Finally, it evaluates the model's performance using various metrics.

Implement K-Means clustering/ hierarchical clustering on sales_data_sample.csv dataset. Determine the number of clusters using the elbow method.

This code is an example of using K-Means clustering to identify clusters in a dataset of sales data. Let's break down the code step by step:

1. **Importing Libraries**: The code imports necessary libraries, including `numpy`, `pandas`, `matplotlib.pyplot`, and `seaborn`.

2. **Loading the Dataset**: It loads the dataset from a CSV file called "sales_data_sample.csv" and specifies the encoding as 'latin1' to read the file properly. The dataset is stored in the `df` DataFrame.

3. **Data Exploration**:
   - `df.head()`: This function displays the first few rows of the dataset to give an overview of its structure.
   - `df.info()`: It provides information about the dataset, including the number of columns, data types, and the presence of any missing values.
   - `df.describe()`: This function generates summary statistics for the numerical columns in the dataset.

4. **Correlation Matrix Visualization**:
   - A correlation matrix is computed using `df.corr()`.
   - The `sns.heatmap` function is used to create a heatmap of the correlation matrix with annotations to visualize the pairwise correlations between columns.
   - The resulting heatmap is displayed using `plt.show()`.

5. **Feature Selection**:
   - The code selects two columns, 'PRICEEACH' and 'MSRP', as the features for clustering. The new DataFrame `df` contains only these two columns.
   - The first few rows of the updated DataFrame are displayed using `df.head()`.

6. **Data Preprocessing**:
   - `df.isna().any()`: This function checks if there are any missing values in the DataFrame. It returns a boolean mask indicating which columns have missing values. In this case, there are no missing values.
   - `df.describe().T`: The summary statistics are provided for the selected columns, but this time, the result is transposed for better readability.
   - `df.shape`: It shows the dimensions of the DataFrame (number of rows and columns).

7. **K-Means Clustering**:
   - The code performs K-Means clustering to identify clusters in the 'PRICEEACH' and 'MSRP' features.
   - It calculates the inertia (within-cluster sum of squares) for different numbers of clusters (from 1 to 10) to help determine the optimal number of clusters.
   - The inertia values are stored in the `inertia` list.

8. **Elbow Method Visualization**:
   - `sns.lineplot`: This function creates a line plot to visualize the inertia values for different numbers of clusters.
   - The x-axis represents the number of clusters, and the y-axis represents the inertia.
   - The plot is displayed using `plt.figure(figsize=(6, 6))`.

9. **K-Means Clustering (Optimal Number of Clusters)**:

- Based on the elbow method, the code selects the optimal number of clusters as 3 (where the "elbow" in the plot occurs). This is done by initializing the `KMeans` model with `n_clusters = 3`.
- `y_kmeans` stores the cluster labels for each data point after fitting the model to the data.
- The scatterplot is created to visualize the clusters in the feature space, and the cluster centers are marked in red.

In summary, the code loads a sales dataset, explores its features, and then performs K-Means clustering to identify clusters in the 'PRICEEACH' and 'MSRP' features. The elbow method is used to determine the optimal number of clusters, and the results are visualized in a scatterplot.