Question1.

Great! It looks like the code is now running successfully, and you've obtained the output for the statistics of the specified columns, as well as the Mean Squared Error (MSE) and R^2 Score for your model.

Summary of the Output:

- **Columns in the Dataset**: The dataset contains the following columns:
- `MedInc`
- `HouseAge`
- `AveRooms`
- `AveBedrms`
- `Population`
- `AveOccup`
- `Latitude`
- `Longitude`
- **Statistics**: The statistics for `AveRooms`, `AveBedrms`, `AveOccup`, and `Population` were printed successfully.
- **Model Evaluation**:
- **Mean Squared Error (MSE)**: 0.5559
- **R^2 Score**: 0.5758

Next Steps:

- **Model Improvement**: If you're looking to improve the model's performance, consider experimenting with different algorithms, feature engineering, or hyperparameter tuning.
- **Data Visualization**: You might want to visualize the results or the relationships between features and the target variable to gain more insights.

3. **Further Analysis**: You can also analyze the residuals or perform cross-validation for a more robust evaluation of your model.

1. **Load the California housing dataset**:

- The code uses `fetch_california_housing()` to load the dataset and creates a DataFrame from it.

```
california_housing = fetch_california_housing()
data = pd.DataFrame(california_housing.data, columns=california_housing.feature_names)
```

- **Provide statistics for the specified columns**:
- The code calculates and prints the statistics for the columns `["AveRooms", "AveBedrms", "AveOccup", "Population"]` using the `describe()` method.

```
statistics = data[["AveRooms", "AveBedrms", "AveOccup", "Population"]].describe()
print(statistics)
```

3. **Split the dataset into training and testing sets**:

- The code splits the dataset into training and testing sets using `train_test_split()`.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
)
```

Question2LR RMSE MAE

Key Changes:

 Importing Metrics: Added mean_absolute_error from sklearn.metrics to calculate MAE.

 Model Evaluation: After making predictions, the code calculates RMSE and MAE:

 RMSE: Calculated using np.sqrt(mean_squared_error(y_test, y_pred)).

 MAE: Calculated using mean_absolute_error(y_test, y_pred).

Explanation of Metrics:

 Root Mean Squared Error (RMSE): Measures the average magnitude of the errors between predicted and actual values, giving higher weight to larger errors.

 Mean Absolute Error (MAE): Measures the average magnitude of the errors in a set of predictions, without considering their direction.

Summary of the Output:

 Statistics for the Specified Columns: The statistics for AveRooms, AveBedrms, AveOccup, and Population were printed successfully.

 Model Evaluation:

 Root Mean Squared Error (RMSE): 0.7456

 Mean Absolute Error (MAE): 0.5332

Interpretation of the Metrics:

 RMSE (0.7456): This value indicates the average error in the predictions made by the model. A lower RMSE value generally indicates a better fit of the model to the data.

 MAE (0.5332): This value represents the average absolute difference between the predicted and actual values. Like RMSE, a lower MAE indicates better model performance.

Question3.

It looks like your code has executed successfully, and you've obtained the shapes of the training and testing sets for the Iris dataset:

 Training set shape: (120, 4) indicates that there are 120 samples and 4 features in the training set.

 Testing set shape: (30, 4) indicates that there are 30 samples and 4 features in the testing set.

Summary:

 The program correctly loaded the Iris dataset, scaled the data, and split it into training and testing sets as per the requirements of your question (Q3).

- **Loading the Iris Dataset**:

```
from sklearn.datasets import load_iris
iris = load_iris()
data = pd.DataFrame(data=iris.data, columns=iris.feature_names)
target = iris.target
```

- This part of the code uses `load_iris()` to load the dataset. The features are stored in `data`, and the target labels are stored in `target`.

- **Scaling the Data**:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data)
```

- Here, `StandardScaler` is used to scale the features to have a mean of 0 and a standard deviation of 1.

3. **Splitting the Dataset**:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(scaled_data, target, test_size=0.2,
random_state=42)
```

- This line splits the scaled data into training and testing sets, with 20% of the data reserved for testing.

Question Train DecisionTree

Explanation of the Code:

1. **Load the Iris Dataset**: The Iris dataset is loaded using load_iris().

2. **Scale the Data**: The features are scaled using StandardScaler().

3. **Split the Dataset**: The scaled data is split into training and testing sets using train_test_split().

4. **Train a Decision Tree Model**: A DecisionTreeClassifier is created and trained on the training data.

 Make Predictions: Predictions are made on the test set.

6. **Evaluate the Model**: The model's performance is evaluated using:

 Accuracy: The proportion of correctly classified instances.

 Precision: The ratio of true positive predictions to the total predicted positives.

 Recall: The ratio of true positive predictions to the total actual positives.

 F1 Score: The harmonic mean of precision and recall.

1. **Train a Decision Tree Model**:

- The code uses `DecisionTreeClassifier` from `sklearn.tree` to create and train the model on the training data.

```
model = DecisionTreeClassifier(random_state=42)
model.fit(X_train, y_train)
```

- **Evaluate Model Performance**:
- The code calculates and prints the following performance metrics:
- **Accuracy**: The proportion of correctly classified instances.
- **Precision**: The ratio of true positive predictions to the total predicted positives.
- **Recall**: The ratio of true positive predictions to the total actual positives.
- **F1 Score**: The harmonic mean of precision and recall.

```
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')
print(f"Accuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1 Score: {f1:.2f}")
```

Conclusion:

The code successfully meets the requirement to train a Decision Tree model and evaluate its performance using the specified metrics

Congratulations! It looks like your Decision Tree model has achieved perfect scores on all evaluation metrics:

- **Accuracy**: 1.00 (100% of the predictions were correct)
- **Precision**: 1.00 (all positive predictions were correct)
- **Recall**: 1.00 (all actual positives were predicted correctly)
- **F1 Score**: 1.00 (the harmonic mean of precision and recall is perfect)

Interpretation:

- **Perfect Performance**: These results indicate that your model is performing exceptionally well on the test dataset. However, it's important to consider the following:
- **Overfitting**: A perfect score can sometimes indicate that the model has overfitted to the training data, especially if the dataset is small or not diverse enough. It's a good idea to validate the model on a separate dataset or use cross-validation to ensure that it generalizes well to unseen data.
- **Dataset Size**: If the dataset is small, the model might memorize the training data rather than learn general patterns.