Code Explanation

We start off by importing the necessary libraries for this project

Step 1: Importing Libraries

```
# import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score, mean_squared_error, mea
n_absolute_error
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import GradientBoostingRegressor
```

- In this step, you import all the necessary libraries for data analysis, visualization, modeling, and evaluation.
- `numpy` and `pandas` are commonly used for data manipulation and analysis.
- `matplotlib.pyplot` and `seaborn` are used for data visualization.
- `train_test_split` from `sklearn.model_selection` is used to split the dataset into training and testing sets.
- `LinearRegression` and `GradientBoostingRegressor` from `sklearn.linear_model` and `sklearn.ensemble` respectively are used to build regression models.
- `r2_score`, `mean_squared_error`, and `mean_absolute_error` from `sklearn.metrics` are used to evaluate model performance.
- `StandardScaler` from `sklearn.preprocessing` is used for feature scaling.

Step 2: Loading the Dataset

```
# load dataset
df = pd.read_csv("C:/Users/Admin/Desktop/Work/LMS projects/in
surance.csv")
df.head()
```

- In this step, you load the dataset using `pd.read_csv()` function from `pandas`.
- Make sure to import the data properly and check on the “foward-slash”, “/”
- The dataset is stored in a DataFrame called `df`.
- You display the first few rows of the dataset using `df.head()` to get an overview of the data structure.

Step 3: Exploratory Data Analysis (EDA)

```
# Check dataset information
df.info()
```

- `df.info()` is used to get information about the dataset including the number of entries, data types, and presence of missing values.

```
# Statistical summary of the dataset
df.describe()
```

- `df.describe()` provides summary statistics (mean, standard deviation, minimum, maximum, etc.) for numerical columns in the dataset.

Step 4: Visualizing Categorical Variables

```
# Define a function to plot a pie chart
def pie_plot(column):
       fig , ax = plt.subplots()
       ax.pie(df[column].value_counts(), autopct="%0.2f%%", l
abels=df[column].value_counts().index)
       ax.set(title=f"Pie Chart of {column}")
```

- This Python function, `pie_plot`, creates a pie chart using matplotlib. It takes a column from a DataFrame (`df`) as input. The pie chart displays the distribution of values in that column. The `autopct="%0.2f%%"` parameter formats the percentage display on the chart. Finally, it sets the title of the chart using the column name.
- The function takes a column name as input and plots a pie chart showing the distribution of categories.

```
# Loop through categorical columns and plot pie charts
columns = ["sex", "smoker", "region"]
for i in columns:
    pie_plot(i)
```

- You loop through the list of categorical columns and call the `pie_plot()` function to plot a pie chart for each column.

Step 5: Visualizing Numerical Variables

```
# Plot histogram of BMI by gender
plt.figure(figsize=(8, 6))
sns.histplot(data=df, x="bmi", hue="sex", kde=True)
plt.title("Distribution of BMI by Gender")
plt.xlabel("BMI")
plt.ylabel("Frequency")
plt.show()
```

- Use `sns.histplot()` from seaborn to plot a histogram of BMI by gender.
- The `hue="sex"` parameter is used to differentiate between male and female.
- The histogram is visualized with a kernel density estimation (KDE) curve.

Step 6: Outlier Detection and Removal

```
# Define a function to remove outliers using the IQR method
def detect_outlier(data, threshold=1.5):
    q1 = np.quantile(data, 0.25)
    q3 = np.quantile(data, 0.75)
    iqr = q3 - q1
    lower_bound = q1 - threshold * iqr
    upper_bound = q3 + threshold * iqr
    return lower_bound, upper_bound
```

- Here, you define a function `detect_outlier()` to detect outliers using the interquartile range (IQR) method.
- The function calculates the lower and upper bounds based on a specified threshold.

```
# Remove outliers from BMI column
low, up = detect_outlier(df["bmi"])
index = df[(df["bmi"] < low) | (df["bmi"] > up)].index
df.drop(index=index, inplace=True)
df.reset_index(drop=True, inplace=True)
```

- Use the `detect_outlier()` function to find the lower and upper bounds for the BMI column.
- Outliers are identified and removed based on these bounds using boolean indexing.

Step 7: Handling Missing Values and Duplicates

```
# Check for missing values
df.isna().sum()
```

- `df.isna().sum()` is used to check for missing values in each column of the dataset.

```
# Check for duplicate rows
df.duplicated().sum()
```

- `df.duplicated().sum()` checks for duplicated rows in the dataset.

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

- Duplicated rows, if any, are removed from the dataset using `df.drop_duplicates()`.

Step 8: Mapping Categorical Variables to Numerical Values

```
# Define mappings for categorical variables
mapper_sex = {"male": 1, "female": 0}
mapper_smoker = {"yes": 1, "no": 0}
mapper_region = {"southeast": 0, "southwest": 1, "northwest":
2, "northeast": 3}

# Apply mappings to categorical columns
df["sex"] = df["sex"].map(mapper_sex)
df["smoker"] = df["smoker"].map(mapper_smoker)
df["region"] = df["region"].map(mapper_region)
```

- Define dictionaries to map categorical variables to numerical values.

- Each dictionary maps categories to corresponding numerical values.
- You apply these mappings to the respective columns in the DataFrame using the `map()` function.

Step 9: Visualizing Relationships and Correlations

```
# Plot heatmap to visualize correlations between features
corr = df.corr()
plt.figure(figsize=(10, 7))
sns.heatmap(corr, annot=True, cmap="Blues")
plt.title("HeatMap", size=20, color="blue")
```

- You can compute the correlation matrix using `df.corr()` to quantify the relationships between numerical features.
- A heatmap is plotted using `sns.heatmap()` from seaborn to visualize the correlations.
- Here you can check if the items are highly correlated which can affect out model
- The `annot=True` parameter adds numerical annotations to the heatmap cells.

Step 10: Model Building and Evaluation

```
# Splitting the dataset into train and test sets
X = np.array(df.iloc[:, :-1])
y = np.array(df["charges"])
X_train, X_test, y_train, y_test = train_test_split(X, y, tes
t_size=0.25, random_state=0)

# Scaling the features using StandardScaler
scaler = StandardScaler().fit(X_train)
```

```
X_train_s = scaler.transform(X_train)
X_test_s = scaler.transform(X_test)
```

- You split the dataset into features (X) and the target variable (y).
- The dataset is divided into training and testing sets using `train_test_split()` from sklearn.
- Features are standardized using `StandardScaler()` to ensure all features are on the same scale.

Step 11: Building Linear Regression Model

```
# Building Linear Regression model
lin_reg = LinearRegression().fit(X_train_s, y_train)

# Making predictions on the test set
pred_lin = lin_reg.predict(X_test_s)

# Evaluating the model
mse_lin = mean_squared_error(y_test, pred_lin)
rmse_lin = np.sqrt(mse_lin)
mae_lin = mean_absolute_error(y_test, pred_lin)
r2_lin = r2_score(y_test, pred_lin)
```

- You train a Linear Regression model (`LinearRegression()`) on the standardized training data (`X_train_s`, `y_train`) using the `fit()` method.
- Predictions are made on the standardized test set (`X_test_s`) using the `predict()` method.
- Evaluation metrics such as Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and R-squared (R2) score are computed using the actual target values (`y_test`) and predicted values (`pred_lin`).

Step 12: Visualizing Model Predictions

```
# Visualizing model predictions
fig, ax = plt.subplots()
ax.scatter(y_test, pred_lin, c="blue")
ax.plot(y_test, y_test, ls="--", color="red")
ax.set(xlabel="Actual Values", ylabel="Predicted Values", tit
le="Evaluating Linear Regression")
ax.grid(axis="both", ls="--", color="gray")
```

- You create a scatter plot to visualize the relationship between actual target values (`y_test`) and predicted values (`pred_lin`) from the Linear Regression model.
- A diagonal line (`y_test = pred_lin`) is plotted for reference to assess how well the predictions align with the actual values.

Step 13: Building Gradient Boosting Regression Model

```
# Building Gradient Boosting Regression model
grb_reg = GradientBoostingRegressor(n_estimators=350, learnin
g_rate=0.05, random_state=0, max_depth=2)
grb_reg.fit(X_train_s, y_train)

# Making predictions on the test set
pred_grb = grb_reg.predict(X_test_s)

# Evaluating the model
mse_grb = mean_squared_error(y_test, pred_grb)
rmse_grb = np.sqrt(mse_grb)
mae_grb = mean_absolute_error(y_test, pred_grb)
r2_grb = r2_score(y_test, pred_grb)
```

- You train a Gradient Boosting Regression model (`GradientBoostingRegressor()`) on the standardized training data (`X_train_s`, `y_train`).
- The hyperparameters such as the number of estimators, learning rate, random state, and maximum depth are specified.
- Predictions are made on the standardized test set (`X_test_s`).
- Evaluation metrics are computed for the Gradient Boosting Regression model similar to the Linear Regression model.

Step 14: Comparing Models

```
# Comparing model performance
rmse = [rmse_grb, rmse_lin]
r2 = [r2_grb, r2_lin]

x = [0, 0.5]

fig, ax = plt.subplots(1, 2, figsize=(10, 4))
ax[0].bar(x, rmse, width=0.15, color=["blue", "green"])
ax[0].set(xticks=[0, 0.5], xticklabels=["Gradient", "Linea
r"], ylabel="RMSE")
ax[1].bar(x, r2, width=0.15, color=["blue", "green"])
ax[1].set(xticks=[0, 0.5], xticklabels=["Gradient", "Linea
r"], ylabel="R2 score")
fig.suptitle("Comparing the Linear and Ensemble models")
```

- You compare the performance of the Linear Regression and Gradient Boosting Regression models using evaluation metrics (RMSE and R-squared score).
- Bar plots are created to visualize the RMSE and R-squared score of both models side by side for comparison.

Step 15: Saving the trained model

```
with open('linear_regression_model.pkl', 'wb') as file:
    pickle.dump(lin_reg, file)

with open('gradient_boosting_model.pkl', 'wb') as file:
    pickle.dump(grb_reg, file)
```

- Here we save our model to our local storage using pickle