**import pandas as pd**

* This line imports the pandas library and gives it the alias pd. You’ll use this to handle and manipulate your dataset, such as reading data from a CSV file or performing data cleaning.

**data = pd.read\_csv('insurance.csv')**

The line of code data = pd.read\_csv('insurance.csv') is used to read a CSV file named insurance.csv into a pandas DataFrame called data.

**#Display Top 5 Rows of The Dataset**

**data.head()**

This function returns the first 5 rows of the DataFrame by default. You can pass an argument to specify a different number of rows (e.g., data.head(10) for the first 10 rows).

**#Check Last 5 Rows of The Dataset**

**data.tail()**

This function returns the last 5 rows of the DataFrame by default. Similar to head(), you can pass an argument to tail() to specify a different number of rows (e.g., data.tail(10) for the last 10 rows).

**#Find Shape of Our Dataset (Number of Rows And Number of Columns)**

**data.shape**

This attribute returns a tuple representing the dimensionality of the DataFrame. The first value in the tuple is the number of rows, and the second value is the number of columns.

**print("Number of Rows", data.shape[0])**

**print("Number of Columns", data.shape[1])**

data.shape[0]: Retrieves the number of rows in the DataFrame.

data.shape[1]: Retrieves the number of columns in the DataFrame.

**#Get Information About Our Dataset Like Total Number Rows, Total Number of Columns, Datatypes of Each Column And Memory Requirement**

**data.info()**

The data.info() method provides a concise summary of your DataFrame, including the number of rows and columns, the data types of each column, and the memory usage.

data.info(): This function gives a summary that includes:

* The total number of rows and columns.
* The data type of each column (e.g., int64, float64, object for strings).
* The number of non-null values in each column (which helps in identifying missing data).
* The memory usage of the DataFrame.

**#Check Null Values In The Dataset**

**data.isnull(). sum()**

The code data.isnull().sum() will check for null (or missing) values in the dataset and provide a count of them for each column.

**#Get Overall Statistics About The Dataset For Numerical Column**

**data.describe () #If You Want Overall Statistics About The Dataset For Numerical Column As Well As Categorical Column Then Use data.describe (include='all')**

data.describe(): This method generates descriptive statistics that summarize the central tendency, dispersion, and shape of a dataset's numerical columns. By default, it only includes numeric types.

data.describe(include='all'): This version includes statistics for both numeric and categorical columns.

**#Covert Columns From String ['sex','smoker', 'region'] To Numerical Values**

**data.head ()**

This function returns the first 5 rows of the DataFrame by default. You can pass an argument to specify a different number of rows (e.g., data.head(10) for the first 10 rows).

**data['sex'].unique()**

The code data['sex'].unique() returns an array of the unique values present in the sex column of the DataFrame. This is useful for understanding what distinct categories or values are present in that column.

**data['sex']=data['sex'].map({'female': 0, 'male':1})**

The code data['sex'] = data['sex'].map({'female': 0, 'male': 1}) is used to convert the categorical values in the sex column to numerical values.

**data.head()**

This function returns the first 5 rows of the DataFrame by default. You can pass an argument to specify a different number of rows (e.g., data.head(10) for the first 10 rows).

**data['smoker']=data['smoker'].map({'yes':1, 'no':0})**

The code data['smoker'] = data['smoker'].map({'yes': 1, 'no': 0}) is used to convert the categorical values in the smoker column into numerical values.

**data.head()**

This function returns the first 5 rows of the DataFrame by default. You can pass an argument to specify a different number of rows (e.g., data.head(10) for the first 10 rows). **data['region'].unique ()**

The code data['region'].unique() retrieves the unique values present in the region column of your DataFrame. This is useful for understanding the distinct categories or regions represented in that column.

**data['region']=data['region'].map({'southwest':1, 'southeast':2, 'northwest': 3, 'northeast':4})**

The code data['region'] = data['region'].map({'southwest': 1, 'southeast': 2, 'northwest': 3, 'northeast': 4}) converts the categorical values in the region column to numerical values.

**data.head()**

This function returns the first 5 rows of the DataFrame by default. You can pass an argument to specify a different number of rows (e.g., data.head(10) for the first 10 rows).

**#Store Feature Matrix In X and Response (Target) In Vector y**

**data.columns**

The code data.columns returns the column names of the DataFrame.

**X = data.drop(['charges'],axis=1)**

The code X = data.drop(['charges'], axis=1) is used to create a new DataFrame X by removing the charges column from the original DataFrame data.

drop(): A pandas method used to remove specified labels from rows or columns.

['charges']: The list of column names to be removed. Here, it's the charges column.

axis=1: Indicates that you are dropping columns (not rows). axis=0 would be used for dropping rows.

**X**

A new DataFrame containing all columns from data except for the charges column.

**y = data['charges']**

The code y = data['charges'] extracts the charges column from the data DataFrame and assigns it to the variable y. This is typically done to separate the target variable from the features in a dataset.

**Y**

This variable will contain the target values (or the dependent variable) that you want to predict.

**#Train/Test split**

**#1. Split data into two part: a training set and a testing set**

**#2. Train the model(s) on training set**

**#3. Test the Model(s) on Testing set**

**from sklearn.model\_selection import train\_test\_split**

The train\_test\_split function from the sklearn.model\_selection module is used to split a dataset into training and testing sets. This is a crucial step in machine learning to evaluate how well your model generalizes to unseen data.

**X\_train, X\_test,y\_train,y\_test=train\_test\_split(X,y,test\_size=0.2,random\_state=42)**

X\_train: Features of the training set.

X\_test: Features of the test set.

y\_train: Target values for the training set.

y\_test: Target values for the test set.

test\_size=0.2: Specifies that 20% of the data will be used for the test set, and the remaining 80% will be used for the training set.

random\_state=42: Ensures that the data split is reproducible. Using the same random\_state value will produce the same split each time you run the code.

**#Import the models**

**from sklearn.linear\_model import LinearRegression**

**from sklearn.svm import SVR**

**from sklearn.ensemble import RandomForestRegressor**

**from sklearn.ensemble import GradientBoostingRegressor**

The code you’ve used imports several regression models from the scikit-learn library:

1. LinearRegression:
   * A linear model that assumes a linear relationship between the input features and the target variable. It's commonly used for predicting continuous values.
2. SVR (Support Vector Regression):
   * A regression model based on Support Vector Machines (SVMs) that tries to fit the error within a certain margin of tolerance. It is useful for capturing non-linear relationships.
3. RandomForestRegressor:
   * An ensemble method that uses multiple decision trees to improve prediction accuracy and control over-fitting. It works well for both linear and non-linear data.
4. GradientBoostingRegressor:
   * An ensemble technique that builds models sequentially to correct the errors of the previous models. It combines weak learners to create a strong predictive model.

**#Model Training**

**lr = LinearRegression()**

**lr.fit(X\_train, y\_train)**

**svm = SVR()**

**svm.fit(X\_train,y\_train)**

**rf = RandomForestRegressor()**

**rf.fit(X\_train, y\_train)**

**gr = GradientBoostingRegressor ()**

**gr.fit(X\_train,y\_train)**

Initialize the Models:

* lr = LinearRegression(): Creates an instance of the Linear Regression model.
* svm = SVR(): Creates an instance of the Support Vector Regression model.
* rf = RandomForestRegressor(): Creates an instance of the Random Forest Regressor model.
* gr = GradientBoostingRegressor(): Creates an instance of the Gradient Boosting Regressor model.

Fit the Models:

* lr.fit(X\_train, y\_train): Trains the Linear Regression model using the training features X\_train and target values y\_train.
* svm.fit(X\_train, y\_train): Trains the Support Vector Regression model using the training features X\_train and target values y\_train.
* rf.fit(X\_train, y\_train): Trains the Random Forest Regressor model using the training features X\_train and target values y\_train.
* gr.fit(X\_train, y\_train): Trains the Gradient Boosting Regressor model using the training features X\_train and target values y\_train.

**#Prediction on Test Data**

**y\_predl = lr.predict(X\_test)**

**y\_pred2 = svm.predict(X\_test)**

**y\_pred3 = rf.predict(X\_test)**

**y\_pred4 = gr.predict(X\_test)**

**df1 = pd.DataFrame ({'Actual':y\_test, 'Lr':y\_predl, 'svm':y\_pred2, 'rf':y\_pred3, 'gr':y\_pred4})**

1. Predictions:
   * lr.predict(X\_test): Predicts target values using the Linear Regression model.
   * svm.predict(X\_test): Predicts target values using the Support Vector Regression model.
   * rf.predict(X\_test): Predicts target values using the Random Forest Regressor model.
   * gr.predict(X\_test): Predicts target values using the Gradient Boosting Regressor model.
2. DataFrame Creation:
   * pd.DataFrame(): Creates a new DataFrame where:
     + 'Actual': Contains the true target values from y\_test.
     + 'Lr': Contains predictions from the Linear Regression model.
     + 'svm': Contains predictions from the Support Vector Regression model.
     + 'rf': Contains predictions from the Random Forest Regressor model.
     + 'gr': Contains predictions from the Gradient Boosting Regressor model.

**df1**

The DataFrame df1 will show a side-by-side comparison of the actual target values and the predictions made by each model. This allows you to visually inspect how well each model is performing.

**#Compare Performance Visually**

**import matplotlib.pyplot as plt**

You can use matplotlib to create visual comparisons of the performance of different models.

**plt.subplot(221)**

**plt.plot(df1['Actual'].iloc[0:11], label='Actual')**

**plt.plot(df1['Lr'].iloc[0:11], label="Lr")**

**plt.legend ()**

**plt.subplot (222)**

**plt.plot(df1['Actual'].iloc[0:11], label='Actual')**

**plt.plot(df1['svm'].iloc[0:11], label="svr")**

**plt.legend()**

**plt.subplot (223)**

**plt.plot(df1['Actual'].iloc[0:11], label='Actual')**

**plt.plot(df1['rf'].iloc[0:11], label="rf")**

**plt.legend()**

**plt.subplot(224)**

**plt.plot(df1['Actual'].iloc[0:11], label='Actual')**

**plt.plot(df1['gr'].iloc[0:11], label="gr")**

**plt.tight\_layout ()**

**plt.legend()**

Your code plots the actual values and predictions from different models on a single figure using matplotlib. Each subplot compares the actual values with the predictions from one model.

**#Evaluating the Algorithm**

**from sklearn import metrics**

To evaluate the performance of your regression models, you can use various metrics available in scikit-learn. Here’s how you can evaluate your models using metrics from the metrics module in scikit-learn.

Common Regression Metrics

1. Mean Absolute Error (MAE): Measures the average magnitude of the errors in a set of predictions, without considering their direction.
2. Mean Squared Error (MSE): Measures the average of the squares of the errors—that is, the average squared difference between the estimated values and the actual value.
3. Root Mean Squared Error (RMSE): The square root of the Mean Squared Error. It provides the magnitude of error in the same units as the target variable.
4. R-squared (Coefficient of Determination): Indicates the proportion of the variance in the dependent variable that is predictable from the independent variables.

**score1 = metrics.r2\_score (y\_test,y\_predl)**

**score2 = metrics.r2\_score (y\_test,y\_pred2)**

**score3 = metrics.r2\_score (y\_test,y\_pred3)**

**score4 = metrics.r2\_score (y\_test,y\_pred4)**

You’re calculating the R-squared (R²) score for each of your models to evaluate their performance. The R² score measures the proportion of the variance in the dependent variable that is predictable from the independent variables. It ranges from 0 to 1, where a higher value indicates a better fit.

metrics.r2\_score(y\_test, y\_predl): Computes the R² score for Linear Regression predictions.

metrics.r2\_score(y\_test, y\_pred2): Computes the R² score for Support Vector Regression predictions.

metrics.r2\_score(y\_test, y\_pred3): Computes the R² score for Random Forest Regressor predictions.

metrics.r2\_score(y\_test, y\_pred4): Computes the R² score for Gradient Boosting Regressor predictions.

Interpretation:

* Higher R² Score: Indicates that the model explains a higher proportion of the variance in the target variable. It generally suggests a better fit.
* Lower R² Score: Indicates that the model explains less of the variance, implying a poorer fit.

Purpose:

* Model Comparison: Helps to compare the predictive power of different models.
* Model Selection: Assists in choosing the best-performing model based on R² scores.

**print (score1, score2, score3, score4)**

print the output of R² Score.

**s1 = metrics.mean\_absolute\_error(y\_test,y\_predl)**

**s2 = metrics.mean\_absolute\_error(y\_test,y\_pred2)**

**s3 = metrics.mean\_absolute\_error(y\_test,y\_pred3)**

**s4 = metrics.mean\_absolute\_error(y\_test,y\_pred4)**

metrics.mean\_absolute\_error(y\_test, y\_predl): Computes the MAE for Linear Regression.

metrics.mean\_absolute\_error(y\_test, y\_pred2): Computes the MAE for Support Vector Regression.

metrics.mean\_absolute\_error(y\_test, y\_pred3): Computes the MAE for Random Forest Regressor.

metrics.mean\_absolute\_error(y\_test, y\_pred4): Computes the MAE for Gradient Boosting Regressor.

Interpretation:

* Lower MAE: Indicates a model's predictions are closer to the actual values, meaning better performance.
* Higher MAE: Indicates larger average errors in the predictions, suggesting poorer performance.

**#Predict Charges For New Customer**

**data ={'age':40,**

**'sex':1,**

**'bmi':40.30,**

**'children':4,**

**'smoker':1,**

**'region':2}**

**df = pd.DataFrame (data, index=[0])**

You’ve created a DataFrame df for a new customer with specified attributes. To predict the charges for this new customer using your trained models

**df**

It seems like you want to view the DataFrame df for the new customer. If you print df, it will display the customer's information in tabular format.

**# Make predictions**

**new\_pred = gr.predict(df)**

**print(new\_pred)**

It looks like you want to make predictions using the Gradient Boosting Regressor (gr) for the new customer data.

**#Save Model Usign Joblib**

**gr = GradientBoostingRegressor ()**

**gr.fit(X,y)**

It looks like you’re re-training your GradientBoostingRegressor model on the entire dataset (X and y).

Initialization:

* gr = GradientBoostingRegressor(): Creates an instance of the GradientBoostingRegressor class.

Training:

* gr.fit(X, y): Fits the model to the entire dataset X (features) and y (target). This trains the model using all available data.

**import joblib**

To use joblib for saving and loading models, you first need to import the library.

**joblib.dump(gr,'model\_joblib\_gr')**

to save the trained model.

**model.predict(df)**

Uses the trained model to predict the target values for the input data in df.

**# GUI**

**from tkinter import \***

Using tkinter, you can create a graphical user interface (GUI) for your model, allowing users to input data and get predictions

**import joblib**

To use joblib for saving and loading models, you first need to import the library.

**def show\_entry():**

**p1 = float(e1.get())**

**p2 = float(e2.get())**

**p3 = float(e3.get())**

**p4 = float(e4.get())**

**p5 = float(e5.get())**

**p6 = float(e6.get())**

**model = joblib.load('model\_joblib\_gr')**

**result = model.predict([[p1,p2,p3,p4,p5,p6]])**

**Label(master, text = "Insurance Cost").grid(row=7)**

**Label(master, text = result).grid(row=8)**

**master =Tk()**

**master.title("Insurance Cost Prediction")**

**label = Label(master, text = "Insurance Cost Prediction", bg = "black",**

**fg ="white").grid(row=0,columnspan=2)**

**Label (master, text = "Enter Your Age").grid(row=1)**

**Label (master, text = "Male Or Female [1/0]").grid(row=2)**

**Label (master, text = "Enter Your BMI Value").grid(row=3)**

**Label (master, text = "Enter Number of Children").grid(row=4)**

**Label (master, text = "Smoker Yes/No [1/0]").grid(row=5)**

**Label (master, text = "Region [1-4]").grid (row=6)**

**e1 = Entry (master)**

**e2 = Entry (master)**

**e3 = Entry (master)**

**e4 = Entry (master)**

**e5 = Entry (master)**

**e6 = Entry (master)**

**e1.grid(row=1,column=1)**

**e2.grid(row=2,column=1)**

**e3.grid(row=3,column=1)**

**e4.grid(row=4,column=1)**

**e5.grid(row=5,column=1)**

**e6.grid(row=6,column=1)**

**Button (master, text="Predict", command=show\_entry).grid()**

**mainloop()**

This code creates a simple GUI for predicting insurance charges based on user input.

