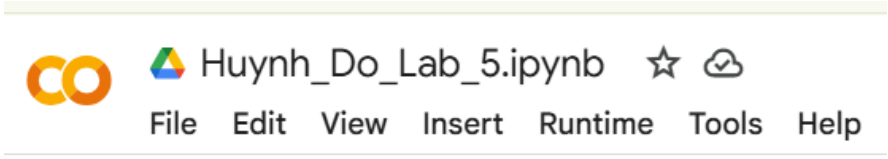
Huynh Do Lab#5:

Objective:

This lab assignment exposes you to categorical variables on your dataset. To use this variable on your predicted model,



1. Import libraries

```
] from google.colab import files
  import pandas as pd
  import numpy as np
  import matplotlib.pyplot as plt
  from sklearn.model_selection import train_test_split
  from sklearn.linear_model import LinearRegression
  from sklearn.preprocessing import OneHotEncoder
  from sklearn.compose import ColumnTransformer
  from sklearn.pipeline import Pipeline
  from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
```

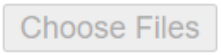
The code above imports several Python libraries commonly used in data analysis.

1. **pandas:** Provides data manipulation and analysis tools using DataFrame structures.
2. **numpy:** Supports numerical operations and matrix calculations.
3. **matplotlib.pyplot:** Creates visualizations, including plots and graphs.
4. **sklearn.model_selection.train_test_split:** Splits data into training and testing sets for model evaluation.
5. **sklearn.linear_model.LinearRegression:** Implements linear regression for modeling numeric relationships.
6. **sklearn.preprocessing.OneHotEncoder:** Converts categorical data into binary-encoded numeric data.
7. **sklearn.compose.ColumnTransformer:** Applies specific preprocessing steps to selected columns.
8. **sklearn.pipeline.Pipeline:** Combines preprocessing and modeling steps into a single workflow.
9. **sklearn.metrics.r2_score:** Measures the proportion of variance explained by the model.
10. **sklearn.metrics.mean_absolute_error:** Calculates the average absolute error between actual and predicted values.
11. **sklearn.metrics.mean_squared_error:** Computes the average squared error, often used to derive RMSE.

2. Upload file insurance.csv

```
# Upload 'insurance.csv' file
uploaded = files.upload()
df = pd.read_csv("insurance.csv")
# Data Summary
print("Data Summary:\n", df.describe())
```

After uploaded



- **insurance.csv**(text/csv) - 55601 bytes, last modified: 4/29/2025 - 100% done

```
Saving insurance.csv to insurance (4).csv
Data Summary:
                age          bmi     children       charges
count  1338.000000  1333.000000  1338.000000   1338.000000
mean     39.207025    30.658545     1.094918  13270.422265
std      14.049960     6.092785     1.205493  12110.011237
min      18.000000    15.960000     0.000000   1121.873900
25%      27.000000    26.315000     0.000000   4740.287150
50%      39.000000    30.400000     1.000000   9382.033000
75%      51.000000    34.675000     2.000000  16639.912515
max      64.000000    53.130000     5.000000  63770.428010
```

3. Process data

Step2: Handle missing values by replacing them with mean (only numeric columns)

```
# Step 2: Handle missing values by replacing them with mean (only numeric columns)
numeric_features = ['age', 'bmi', 'children', 'charges']
df[numeric_features] = df[numeric_features].fillna(df[numeric_features].mean())
```

- The method replaces missing values (NaN) in each column with the mean value of that column:
- df[numeric_features].mean() calculates the mean for each specified column and applies it as the replacement for missing values.

- The mean is a numerical calculation, so it is only applicable to numeric columns.

Step 3: Separate features (X) and target variable (y)

```
# Step 3: Separate features (X) and target variable (y)
X = df.drop('charges', axis=1)
y = df['charges']
```

- **Features (X):** All columns except the target variable (charges). These are the predictors used to estimate the target value.
- **Target Variable (y):** The charges column, which is the value being predicted based on the input features.

Step 4: Set up OneHotEncoder for categorical variables

```
# Step 4: Set up OneHotEncoder for categorical variables
categorical_features = ['sex', 'smoker', 'region']

preprocessor = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(drop='first'), categorical_features)
    ],
    remainder='passthrough'
)
```

To converts categorical variables (sex, smoker, region) into numerical form using OneHotEncoder.

- Machine learning models require numerical input, so categorical data must be encoded.
- OneHotEncoder creates binary (0/1) columns for each category, effectively transforming each categorical variable into multiple columns.
- **Drop First:** The drop='first' parameter prevents multicollinearity by excluding the first category, reducing the number of columns without losing information.

Step 5: Create a pipeline: Preprocessing + Linear Regression

```
# Step 5: Create a pipeline: Preprocessing + Linear Regression
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', LinearRegression())
])
```

To constructs a pipeline that streamlines data processing and model training in a single workflow.

- **Pipeline:** The pipeline consists of two main steps — preprocessing and regression.
  - **Preprocessing:** Applies the OneHotEncoder to categorical variables, converting them into numerical columns.
  - **Regression:** Implements Linear Regression to predict the target variable (charges) based on the processed features.
- **Pipeline Structure** will ensure that the data processing and model training steps are executed sequentially and consistently, preventing data leakage and maintaining the integrity of the model training process.

Step 6: Split the dataset (70% train, 30% test)

```
# Step 6: Split the dataset (70% train, 30% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

To split the dataset into training and testing subsets.

- **Data Split:** The dataset is divided as follows:
  - **70% Training Data:** Used to train the regression model.
  - **30% Testing Data:** Used to evaluate the model's predictive accuracy.
- **Why 70/30?** This split ratio is common practice, providing enough data for training while reserving a significant portion for testing.
- **Random State:** A fixed seed ensures reproducibility, meaning the split will be the same each time the code is run, allowing for consistent evaluation.

Step 7: Fit the pipeline model

```
# Step 7: Fit the pipeline model and predict
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)
```

The model using the training data and makes predictions on the test data.

- **Model Training:** The pipeline, which includes data preprocessing and linear regression, is applied to the training data. This step fits the model to identify the relationship between input features and the target variable (charges).
- **Prediction:** Once the model is trained, it is used to predict charges based on the test dataset. This provides a set of predicted values (y_pred) that can be compared against the actual test values (y_test).

Step 8: Evaluate the model and adjust Adjusted R^2 Calculation

```
# Step 8: Evaluate the model and adjust Adjusted R² Calculation
r2 = r2_score(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
n = len(y_test)
p = len(X_train.columns)
adjusted_r2 = 1 - (1 - r2) * (n - 1) / (n - p - 1)
```

To evaluate the model's performance using multiple metrics to assess accuracy and effectiveness.

- **R^2 Score:** Measures how well the model's predictions align with actual values. A score of 1 indicates perfect predictions, while 0 indicates no predictive power.
- **Adjusted R^2:** Adjusts the R^2 score to account for the number of predictors. It penalizes the model for including irrelevant features, providing a more accurate measure of model performance when multiple features are involved.
- **Mean Absolute Error (MAE):** Calculates the average absolute difference between predicted and actual values, indicating how far off predictions are, on average.
- **Root Mean Squared Error (RMSE):** Measures the square root of the average squared differences between predicted and actual values, emphasizing larger errors.

Step 9: Extract model coefficients and intercept:

```
# Step 9: Extract model coefficients and intercept
regressor = pipeline.named_steps['regressor']
onehot_feature_names = pipeline.named_steps['preprocessor'].named_transformers_['cat'].get_feature_names_out(categorical_features)
numerical_features = [col for col in X.columns if col not in categorical_features]
all_feature_names = np.concatenate([onehot_feature_names, numerical_features])
```

To extract the model coefficients and intercept to understand the impact of each feature on the prediction.

- **Model Coefficients:** These values represent the weight assigned to each feature in the linear regression model. A positive coefficient indicates a positive relationship with the target (charges), while a negative coefficient indicates a negative relationship.
- **Intercept:** This is the baseline value of the target variable when all feature values are zero. It serves as the starting point for the regression line.
- **Purpose:** Extracting coefficients and the intercept helps in interpreting the model by quantifying how each feature influences the predicted charges. It provides insight into the relative importance and direction of each predictor.

Step 10: Create a summary table and model intercept

```
#Step 10:  Create a summary table and model intercept
coefficients_table = pd.DataFrame({
    'Feature': all_feature_names,
    'Coefficient': regressor.coef_
})
```

1. **Summary Table Creation:**
   - A DataFrame is created to list each feature along with its corresponding coefficient.
   - This provides a clear, organized view of how each feature influences the target variable (charges).
   - Features include both the encoded categorical variables and the original numerical variables.
2. **Intercept Calculation:**
   - The intercept represents the baseline value of the target (charges) when all feature values are zero.
   - It serves as the starting point for the regression line and is essential for interpreting the model's output.

- **Step 11**: Output Results

```
# Step 11: Output Results
print("R^2 Score:", round(r2, 4))
print("Adjusted R^2 Score:", round(adjusted_r2, 4))
print("Model Intercept:", round(intercept, 4))
print("Mean Absolute Error (MAE):", round(mae, 4))
print("Root Mean Squared Error (RMSE):", round(rmse, 4))
print("\nModel Coefficients:")
print(coefficients_table)
```

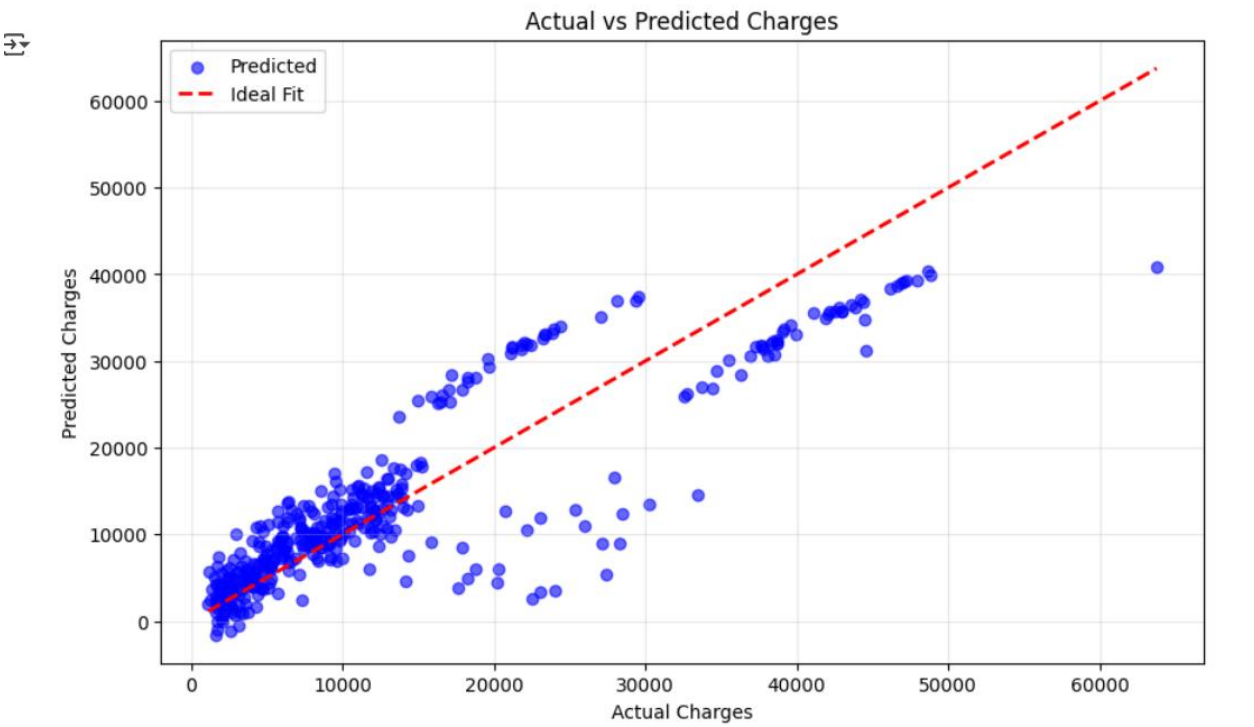
```
R^2 Score: 0.7696
Adjusted R^2 Score: 0.7661
Model Intercept: -12124.1334
Mean Absolute Error (MAE): 4131.5105
Root Mean Squared Error (RMSE): 5811.6998

Model Coefficients:
            Feature   Coefficient
0          sex_male    125.769487
1        smoker_yes  23636.893270
2  region_northwest   -479.197817
3  region_southeast   -892.679970
4  region_southwest   -888.777642
5               age    261.929552
6               bmi    338.732514
7          children    419.370610
```

- **Step 12**: Plot Actual vs Predicted

```python
# Step 12: Plot Actual vs Predicted
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred, alpha=0.6, c='blue', label='Predicted')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], '--r', linewidth=2, label='Ideal Fit')
plt.title('Actual vs Predicted Charges')
plt.xlabel('Actual Charges')
plt.ylabel('Predicted Charges')
plt.legend()
plt.grid(alpha=0.3)
plt.show()
```

1. **Scatter Plot Creation:**
   - A scatter plot is generated with the actual values (y_test) on the x-axis and the predicted values (y_pred) on the y-axis.
   - Each point represents a test data instance, showing how closely the predicted values align with the actual values.
2. **Ideal Fit Line:**
   - A reference line (--r) is plotted to indicate the ideal 1:1 relationship where predicted values perfectly match actual values.
   - The closer the points are to this line, the more accurate the model's predictions.
3. **Plot Aesthetics:**
   - Gridlines are added for clarity.
   - The plot is labeled appropriately with titles and axes to indicate the meaning of each axis.



Interpretation:

1. **R^2 Score Analysis:**
   - The R^2 score of **0.7696** indicates that approximately **76.96% of the variance in charges** can be explained by the model. This suggests a fairly strong fit, but there is still room for improvement.
2. **Adjusted R^2 Analysis:**
   - The Adjusted R^2 score of **0.7661** slightly decreases from the R^2 score, accounting for the number of predictors. This minor decrease indicates that the added predictors are still contributing positively to the model without introducing too much noise.
3. **Error Analysis:**
   - The **Mean Absolute Error (MAE)** of **4131.51** indicates that, on average, the model's predictions are off by about **$4131.51** from the actual charges.
   - The **Root Mean Squared Error (RMSE)** of **5811.70** further emphasizes that the model's prediction errors are relatively high, suggesting some outliers or potential non-linearity in the data.
4. **Intercept Analysis:**
   - The intercept of **-12124.13** represents the baseline charge when all feature values are zero. This is not a realistic scenario but provides a reference point for the regression equation.
5. **Plot Analysis:**
   - The scatter plot shows a general positive trend, indicating that higher actual charges generally correspond to higher predicted charges.

Conclusion:

The linear regression model demonstrates a moderately strong predictive capability, explaining approximately **77% of the variance in insurance charges**. However, the relatively high RMSE of **$5811.70** and MAE of **$4131.51** indicate substantial prediction errors, suggesting that further optimization may be necessary. The model shows a generally positive relationship between the predictors and insurance charges, but the variability in predictions highlights potential areas for refinement, such as handling outliers or incorporating non-linear features.