```
Start coding or generate with AI.
```

```
import os

zip_file_path = "/content/archive (2).zip"

# Ensure the file exists before attempting tc
if os.path.exists(zip_file_path):
    !unzip "{zip_file_path}"
else:
    print(f"Error: The file {zip_file_path} w
```

```
Archive:  /content/archive (2).zip
  inflating: insurance.csv
```

Load Dataset

Subtask:

Load the unzipped dataset into a pandas DataFrame. The file will likely be named 'insurance.csv' or similar after unzipping.

Reasoning: To load the dataset into a pandas DataFrame, I need to import the pandas library and use its `read_csv` function. Then, I will display the first few rows to confirm successful loading.

```
import pandas as pd

df = pd.read_csv('insurance.csv')
print("DataFrame 'df' loaded successfully. Displaying the first 5 rows:")
df.head()
```

```
DataFrame 'df' loaded successfully. Displaying the first 5 rows:
```

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

Next steps: Generate code with df New interactive sheet

Preprocess Data

Subtask:

Encode categorical columns ('sex', 'smoker', 'region') using one-hot encoding. Select the features ('age', 'bmi', 'children', 'smoker') and the target ('charges') as specified.

Reasoning: To preprocess the data, I will first apply one-hot encoding to the specified categorical columns ('sex', 'smoker', 'region') using `pd.get_dummies()`. Then, I will select the requested features ('age', 'bmi', 'children', 'smoker') for `X` and 'charges' for `y` from the processed DataFrame.

```
df_encoded = pd.get_dummies(df, columns=['sex', 'smoker', 'region'], drop_first

X = df_encoded[['age', 'bmi', 'children', 'smoker_yes']]
y = df_encoded['charges']

print("Shape of features (X):", X.shape)
print("Shape of target (y):", y.shape)
print("First 5 rows of X:")
print(X.head())
print("First 5 rows of y:")
print(y.head())
```

```
Shape of features (X): (1338, 4)
Shape of target (y): (1338,)
First 5 rows of X:
   age     bmi  children  smoker_yes
0   19  27.900         0        True
1   18  33.770         1       False
2   28  33.000         3       False
```

```
3   33  22.705         0       False
4   32  28.880         0       False
First 5 rows of y:
0    16884.92400
1     1725.55230
2     4449.46200
3    21984.47061
4     3866.85520
Name: charges, dtype: float64
```

Split Data

Subtask:

Split the preprocessed data into training and testing sets to evaluate model performance.

Reasoning: To split the preprocessed data into training and testing sets, I need to import the `train_test_split` function from `sklearn.model_selection` and apply it to `X` and `y` with the specified `test_size` and `random_state`. Afterwards, I will print the shapes of the resulting datasets to confirm the split.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random

print("Shape of X_train:", X_train.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of y_test:", y_test.shape)
```

```
Shape of X_train: (1070, 4)
Shape of X_test: (268, 4)
Shape of y_train: (1070,)
Shape of y_test: (268,)
```

Train Linear Regression

Subtask:

Train a standard Linear Regression model on the training data.

Reasoning: To train a Linear Regression model, I need to import the `LinearRegression` class, instantiate it, and then fit it to the training data (`X_train` and `y_train`).

```
from sklearn.linear_model import LinearRegression

# Instantiate the Linear Regression model
linear_model = LinearRegression()

# Fit the model to the training data
linear_model.fit(X_train, y_train)

print("Linear Regression model trained successfully.")
```

```
Linear Regression model trained successfully.
```

Train Ridge Regression Models

Subtask:

Train several Ridge Regression models with alpha values 0.1, 1, 10, and 100 on the training data.

Reasoning: To train Ridge Regression models, I need to import the `Ridge` class, define the specified alpha values, and then iterate through these alphas to instantiate and fit each Ridge model to the training data.

```
from sklearn.linear_model import Ridge

alphas = [0.1, 1, 10, 100]
ridge_models = {}

for alpha in alphas:
    ridge_model = Ridge(alpha=alpha)
    ridge_model.fit(X_train, y_train)
    ridge_models[f'Ridge_alpha_{alpha}'] = ridge_model
    print(f"Ridge Regression model with alpha={alpha} trained successfully.")

print("All Ridge Regression models trained successfully and stored in 'ridge_mc
```

```
Ridge Regression model with alpha=0.1 trained successfully.
Ridge Regression model with alpha=1 trained successfully.
Ridge Regression model with alpha=10 trained successfully.
Ridge Regression model with alpha=100 trained successfully.
All Ridge Regression models trained successfully and stored in 'ridge_models' di
```

Evaluate Model Performance

Subtask:

Evaluate the performance of the trained Linear Regression model and all Ridge Regression models on the test data. Use appropriate metrics like R-squared and Mean Absolute Error (MAE).

Reasoning: The previous steps successfully trained all required models. Now, I need to evaluate the performance of these models on the test data. To do this efficiently, I will write a function that takes a model, makes predictions on the test set, and calculates key metrics like R-squared and MAE. I will then apply this function to the Linear Regression model and each Ridge Regression model.

```
from sklearn.metrics import r2_score, mean_ab

results = {}

def evaluate_model(model, X_test, y_test, moc
    y_pred = model.predict(X_test)
    r2 = r2_score(y_test, y_pred)
    mae = mean_absolute_error(y_test, y_pred)
    print(f"{model_name}:\n  R-squared: {r2:.
    return {'R-squared': r2, 'MAE': mae}

# Evaluate Linear Regression model
results['Linear_Regression'] = evaluate_model

# Evaluate Ridge Regression models
for name, model in ridge_models.items():
    results[name] = evaluate_model(model, X_t

print("All models evaluated successfully. Res
```

```
Linear Regression:
  R-squared: 0.7811
  MAE: 4213.80

Ridge_alpha_0.1:
  R-squared: 0.7811
  MAE: 4215.00

Ridge_alpha_1:
  R-squared: 0.7809
  MAE: 4225.75

Ridge_alpha_10:
  R-squared: 0.7773
  MAE: 4330.80

Ridge_alpha_100:
  R-squared: 0.6888
  MAE: 5204.42

All models evaluated successfully. Results stored in 'results' dictionary.
```

Compare and Visualize Models

Subtask:

Identify the best performing Ridge Regression alpha, visualize the comparison of all models, and summarize the findings.

Reasoning: To compare the models effectively and prepare for visualization, I will convert the `results` dictionary, which contains performance metrics for all models, into a pandas DataFrame. This will provide a structured overview of each model's R-squared and MAE scores.

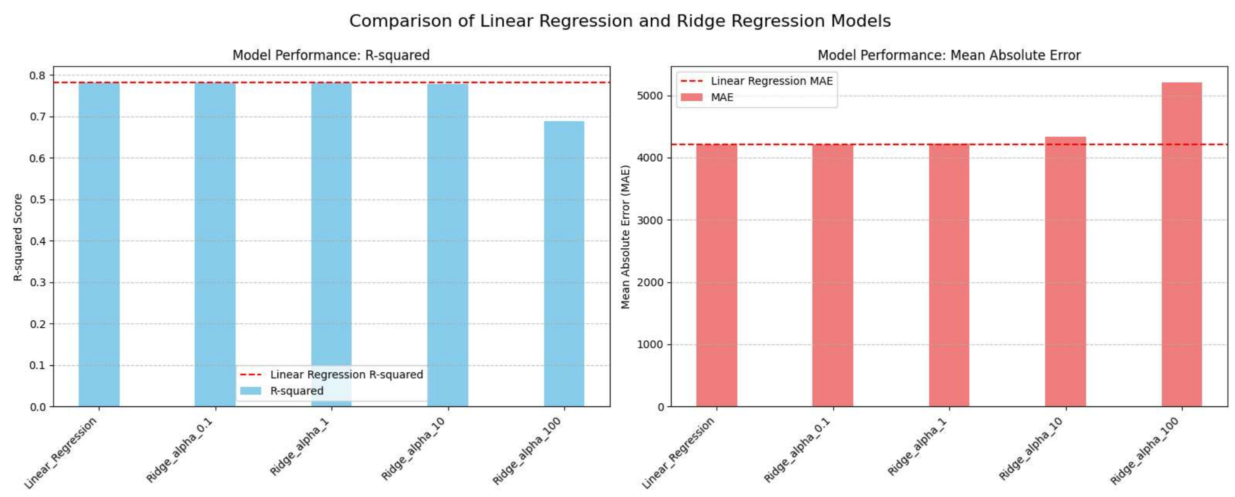
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Convert results dictionary to a DataFrame
results_df = pd.DataFrame(results).T
results_df.index.name = 'Model'

print("Model performance results:")
print(results_df)
```

```
Model performance results:
                   R-squared          MAE
Model
Linear_Regression   0.781115  4213.798595
Ridge_alpha_0.1     0.781092  4214.995041
Ridge_alpha_1       0.780872  4225.750279
Ridge_alpha_10      0.777258  4330.796360
Ridge_alpha_100     0.688764  5204.424227
```

Reasoning: Now that the performance results are in a DataFrame, I will identify the best performing Ridge Regression model based on R-squared and MAE, then visualize the R-squared and MAE scores for all models using bar plots to facilitate comparison.

```
ridge_results_df = results_df[results_df.index.str.startswith('Ridge')]
```

```
best_ridge_r2 = ridge_results_df['R-squared'].idxmax()
best_ridge_mae = ridge_results_df['MAE'].idxmin()

print(f"Best Ridge Regression model by R-squared: {best_ridge_r2} (R-squared: {
print(f"Best Ridge Regression model by MAE: {best_ridge_mae} (MAE: {ridge_resul

# Prepare data for plotting
models = results_df.index
r2_scores = results_df['R-squared']
mae_scores = results_df['MAE']

x = np.arange(len(models))  # the label locations
width = 0.35  # the width of the bars

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))

# R-squared plot
ax1.bar(x, r2_scores, width, label='R-squared', color='skyblue')
ax1.set_ylabel('R-squared Score')
ax1.set_title('Model Performance: R-squared')
ax1.set_xticks(x)
ax1.set_xticklabels(models, rotation=45, ha="right")
ax1.axhline(y=results_df.loc['Linear_Regression', 'R-squared'], color='r', line
ax1.legend()
ax1.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()

# MAE plot
ax2.bar(x, mae_scores, width, label='MAE', color='lightcoral')
ax2.set_ylabel('Mean Absolute Error (MAE)')
ax2.set_title('Model Performance: Mean Absolute Error')
ax2.set_xticks(x)
ax2.set_xticklabels(models, rotation=45, ha="right")
ax2.axhline(y=results_df.loc['Linear_Regression', 'MAE'], color='r', linestyle=
ax2.legend()
ax2.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()

plt.suptitle('Comparison of Linear Regression and Ridge Regression Models', for
plt.show()

# Summarize findings
print("\n--- Summary of Findings ---")
print("1. The Linear Regression model and Ridge Regression model with alpha=0.1
print("2. As the alpha value for Ridge Regression increases (from 0.1 to 100),
print("3. Specifically, Ridge_alpha_0.1 is the best performing Ridge model, clc
print("4. Ridge Regression with higher regularization (alpha=100) shows much lc
Best Ridge Regression model by R-squared: Ridge_alpha_0.1 (R-squared: 0.7811)
Best Ridge Regression model by MAE: Ridge_alpha_0.1 (MAE: 4215.00)
```



Final Task

Subtask:

Summarize the findings, including the performance of Linear Regression, the best performing Ridge Regression model, its optimal alpha value, and any insights gained from the comparison.

Summary:

Data Analysis Key Findings

- The Linear Regression model achieved an R-squared score of approximately 0.7811 and a Mean Absolute Error (MAE) of $4213.80 on the test data.
- Ridge Regression with an alpha value of 0.1 performed very similarly to Linear Regression, yielding an R-squared of approximately 0.7811 and an MAE of $4215.00. This indicates that for this dataset and features, a small amount of regularization did not significantly alter performance.
- The best performing Ridge Regression model, based on both R-squared (highest) and MAE (lowest), was `Ridge_alpha_0.1`.
- As the alpha value for Ridge Regression increased, the model's performance consistently degraded.

  - `Ridge_alpha_1` showed a slight decrease in R-squared (0.7809) and increase in MAE ($4225.75).
  - `Ridge_alpha_10` resulted in a notable drop in R-squared (0.7773) and rise in MAE ($4330.80).
  - `Ridge_alpha_100` exhibited significantly worse performance with an R-squared of approximately 0.6888 and a much higher MAE of $5204.42, indicating substantial underfitting.

Insights or Next Steps

- For this specific dataset and chosen features, regularization does not appear to offer a significant advantage over standard Linear Regression, as the model with the lowest effective regularization (alpha=0.1) performed almost identically. This suggests that the initial feature set might not suffer from severe multicollinearity or overfitting to the extent that strong regularization is beneficial.
- Further analysis could involve exploring why higher alpha values lead to such a drastic performance drop, confirming if it's due to penalizing truly important features too heavily, or if the chosen features have limited noise to begin with.