```
from google.colab import files
uploaded = files.upload()
```

Choose files housing.csv

- **housing.csv**(text/csv) - 1423529 bytes, last modified: 22/09/2019 - 100% done

```
Saving housing.csv to housing.csv
```

```
import os
os.listdir()
```

```
['.config', 'housing.csv', 'sample_data']
```

```
import pandas as pd

df = pd.read_csv("housing.csv")
df.head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	med
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	

Next steps: Generate code with df | New interactive sheet

```
df.shape
```

```
(20640, 10)
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object 
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

```
df.isnull().sum()
```

	0
longitude	0
latitude	0
housing_median_age	0
total_rooms	0
total_bedrooms	207
population	0
households	0
median_income	0
median_house_value	0
ocean_proximity	0

dtype: int64

The target variable is median_house_value, which is continuous, making this a regression problem. Initial inspection shows missing values in the total_bedrooms column, which need to be handled before training models.

```
median_bedrooms = df['total_bedrooms'].median()
df['total_bedrooms'].fillna(median_bedrooms, inplace=True)
```

```
/tmp/ipython-input-61466885.py:2: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series t
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on wh

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)'


  df['total_bedrooms'].fillna(median_bedrooms, inplace=True)
```

```
df.isnull().sum()
```

	0
longitude	0
latitude	0
housing_median_age	0
total_rooms	0
total_bedrooms	0
population	0
households	0
median_income	0
median_house_value	0
ocean_proximity	0

dtype: int64

The total_bedrooms feature contained missing values. These were handled using median imputation.

```
df_encoded = pd.get_dummies(df, columns=['ocean_proximity'])
```

```
df_encoded.head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms
0	-122.23	37.88	41.0	880.0	129.0
1	-122.22	37.86	21.0	7099.0	1106.0
2	-122.24	37.85	52.0	1467.0	190.0
3	-122.25	37.85	52.0	1274.0	235.0
4	-122.25	37.85	52.0	1627.0	280.0

Next steps: Generate code with df_encoded | New interactive sheet

```
df_encoded.shape
```

```
(20640, 14)
```

The categorical feature ocean_proximity was converted into numerical form

```
X = df_encoded.drop('median_house_value', axis=1)
y = df_encoded['median_house_value']
```

```
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_state=42
)
```

The dataset is split into training and testing sets.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Model 1:Linear Regression

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error
import numpy as np

lin_reg = LinearRegression()
lin_reg.fit(X_train_scaled, y_train)
```

```
▾ LinearRegression
LinearRegression()
```

```
# Predictions
y_train_pred = lin_reg.predict(X_train_scaled)
y_test_pred = lin_reg.predict(X_test_scaled)

# Errors
rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))
mae_test = mean_absolute_error(y_test, y_test_pred)

rmse_train, rmse_test, mae_test
```

```
(np.float64(68433.93736666226),
 np.float64(70060.52184473518),
 50670.73824097191)
```

```
results = []

results.append({
    "Model": "Linear Regression",
    "RMSE (Train)": rmse_train,
    "RMSE (Test)": rmse_test,
    "MAE (Test)": mae_test
})
```

```
results = []
```

```
results.append({
    "Model": "Linear Regression",
    "RMSE (Train)": rmse_train,
    "RMSE (Test)": rmse_test,
    "MAE (Test)": mae_test
})
```

```
import pandas as pd

results_df = pd.DataFrame(results)
results_df
```

	Model	RMSE (Train)	RMSE (Test)	MAE (Test)
0	Linear Regression	68433.937367	70060.521845	50670.738241

Model 2: Ridge Regression

```
from sklearn.linear_model import Ridge

ridge_reg = Ridge(alpha=1.0)
ridge_reg.fit(X_train_scaled, y_train)
```

```
▾ Ridge
Ridge()
```

```
# Predictions
y_train_pred_ridge = ridge_reg.predict(X_train_scaled)
y_test_pred_ridge = ridge_reg.predict(X_test_scaled)

# Errors
rmse_train_ridge = np.sqrt(mean_squared_error(y_train, y_train_pred_ridge))
rmse_test_ridge = np.sqrt(mean_squared_error(y_test, y_test_pred_ridge))
mae_test_ridge = mean_absolute_error(y_test, y_test_pred_ridge)

rmse_train_ridge, rmse_test_ridge, mae_test_ridge
```

```
(np.float64(68433.94489002795),
 np.float64(70057.43221282726),
 50668.1315638925)
```

```
results.append({
    "Model": "Ridge Regression",
    "RMSE (Train)": rmse_train_ridge,
    "RMSE (Test)": rmse_test_ridge,
    "MAE (Test)": mae_test_ridge
})
```

```
import pandas as pd
pd.DataFrame(results)
```

	Model	RMSE (Train)	RMSE (Test)	MAE (Test)
0	Linear Regression	68433.937367	70060.521845	50670.738241

Model 3 — Decision Tree Regressor

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(random_state=42)
tree_reg.fit(X_train, y_train)
```

```
DecisionTreeRegressor(random_state=42)
```

```
# Predictions
y_train_pred_tree = tree_reg.predict(X_train)
y_test_pred_tree = tree_reg.predict(X_test)

# Errors
rmse_train_tree = np.sqrt(mean_squared_error(y_train, y_train_pred_tree))
rmse_test_tree = np.sqrt(mean_squared_error(y_test, y_test_pred_tree))
mae_test_tree = mean_absolute_error(y_test, y_test_pred_tree)

rmse_train_tree, rmse_test_tree, mae_test_tree
```

```
(np.float64(0.0), np.float64(69038.97147031713), 43577.561046511626)
```

```
results.append({
    "Model": "Decision Tree Regressor",
    "RMSE (Train)": rmse_train_tree,
    "RMSE (Test)": rmse_test_tree,
    "MAE (Test)": mae_test_tree
})
```

```
import pandas as pd

results_df = pd.DataFrame(results)
results_df
```

	Model	RMSE (Train)	RMSE (Test)	MAE (Test)
0	Linear Regression	68433.937367	70060.521845	50670.738241
1	Ridge Regression	68433.944890	70057.432213	50668.131564
2	Decision Tree Regressor	0.000000	69038.971470	43577.561047

Next steps: Generate code with `results_df` New interactive sheet

Underfitting and Overfitting

In this experiment, Linear Regression showed high training and testing RMSE values that were close to each other, indicating underfitting due to high bias. The model was too simple to capture the non-linear relationships present in housing price data. The Decision Tree Regressor achieved very low training error but significantly higher testing error, which is a clear sign of overfitting caused by high variance and excessive model complexity. Ridge Regression reduced overfitting by applying regularization, resulting in better generalization performance compared to the baseline linear model. This comparison demonstrates the bias–variance trade-off in regression models.

A key real-world machine learning issue in this dataset is **non-linearity**