📶 California Housing Price Prediction

Final project for **Python Machine Learning Diploma** 📊 📈

This project was deployed and Hosted in Streamlit to provide an App For interactive California Housing Price Predicting.

Click Here To Visit California Housing Price Predictor App!

This dataset is from Kaggle and is used in this notebook for educational purposes. Click here to go to the original dataset.

Instructor: **Prof. Mostafa Othman**

By: **Sawsan Abdulbari**

Table of contents

1. **Introduction** to the Project

Context:

California Housing dataset is a great dataset to be utilized for implementing machine learning algorithms as it requires data cleaning, has a clear but varying list of variables and is an optimal size for building a medium sized ML model.

Content:

The dataset consist of information from the 1990 California census, with following columns providing insights:

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

Acknowledgements:

This dataset was also used in 'Hands-On Machine learning with Scikit-Learn and TensorFlow' by Aurélien Géron.

The data was initially featured:

Pace, R. Kelley, and Ronald Barry. "Sparse spatial autoregressions." Statistics & Probability Letters 33.3 (1997): 291-297.

2. Getting **Data** and **Imports**

2.1. Importing **Libraries** and **Modules**

```
# Data handling
import numpy as np
import pandas as pd

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns  # Enhances visualizations

# Preprocessing and Pipelines
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.utils import shuffle

# Feature Selection
# Models
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

# Validation
from sklearn.model_selection import cross_validate

# Evaluation Metrics
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error

# Saving
from joblib import dump
```

2.2. Loading the **Dataset**

```
df = pd.read_csv('/content/housing.csv')
```

```
# Displaying the first few rows of the DataFrame for a quick overview
df.head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	household
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.

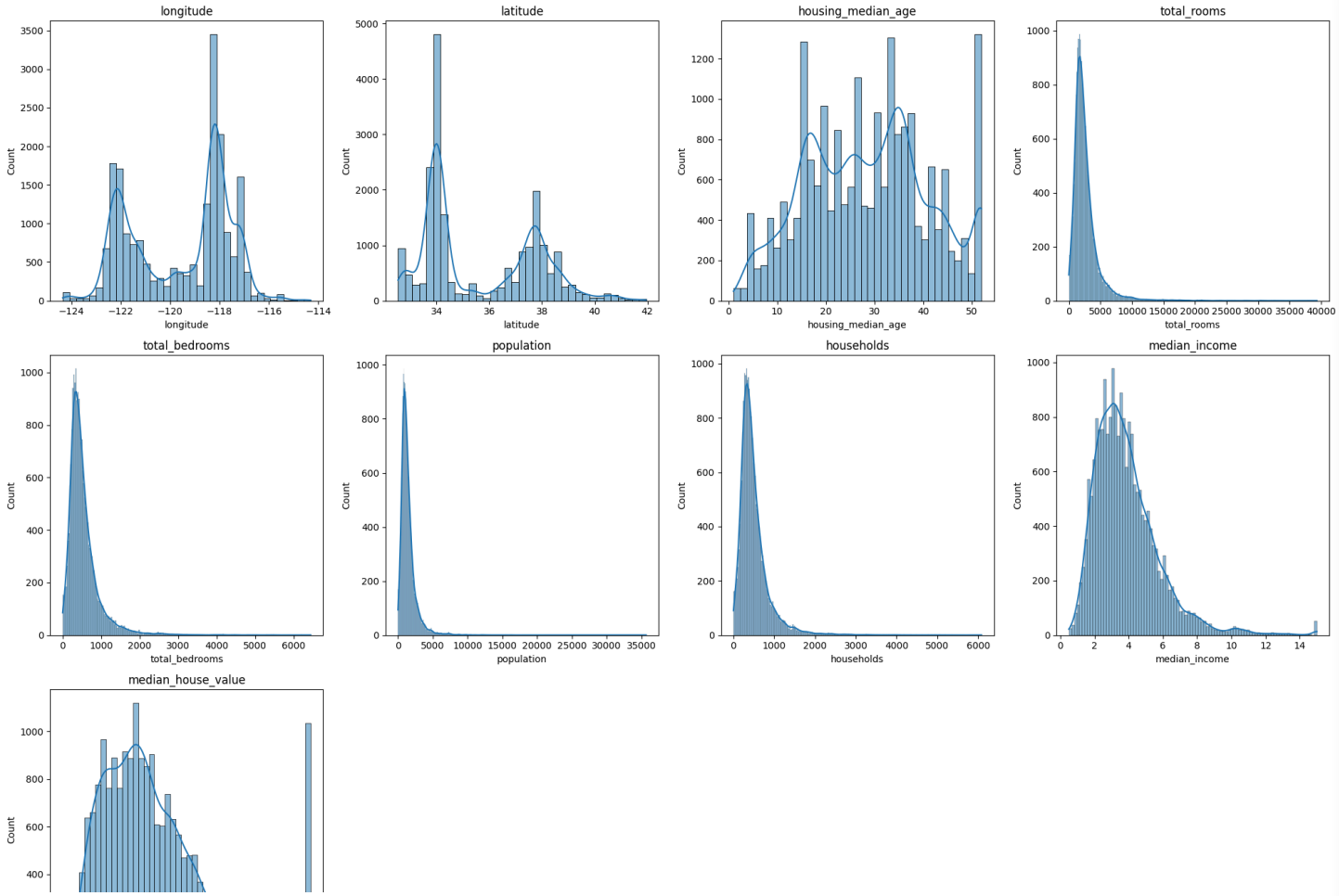
Next steps: View recommended plots

3. **Understanding** the Data

```
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
```

```
# Creating a histograms for numerical features
plt.figure(figsize=(20, 15))
for i, col in enumerate(df.columns[:-1]):  # Excluding the last column (ocean_proximity)
    plt.subplot(3, 4, i + 1)  # Adjusting the subplot grid to accommodate 10 columns
    sns.histplot(df[col], kde=True)
    plt.title(col)
plt.tight_layout()
plt.show()
```




```
# Displaying the shape of the dataset
print("Dataset shape:", df.shape)
```

```
Dataset shape: (20640, 10)
```

3.1. Is there **Duplicated** Data?

[ ] ↳ *1 cell hidden*

3.2. Are there **Missing** Values in the Data?

[ ] ↳ *1 cell hidden*

3.3. Are there **Outlier** Values in the Data?

[ ] ↳ *2 cells hidden*

4. Exploratory Data Analysis (**EDA**)

```
len(df)
```

```
20640
```

```
df.columns
```

```
Index(['longitude', 'latitude', 'housing_median_age', 'total_rooms',
       'total_bedrooms', 'population', 'households', 'median_income',
       'median_house_value', 'ocean_proximity'],
      dtype='object')
```

```
df['ocean_proximity'].value_counts()
```

```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

4.1. **Statistical** Analysis with **Visualizations**

[ ] ↳ *1 cell hidden*

4.2. What is the **correlation** between **numerical columns**?

[ ] ↳ *1 cell hidden*

4.3. **Overview** of **relationships** between the **variables**.

[ ] ↳ *1 cell hidden*

4.4. **Statistical Summary**

```
# Summarizing statistics for the dataset
print("\nStatistics summary:")
df.describe()
```

Statistics summary:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	populatio
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.0000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.4767
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.4621
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.0000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.0000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.0000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.0000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.0000

```
df.describe().round(2).T # Providing a statistical summary of the numerical columns
```

	count	mean	std	min	25%	50%	75%	
longitude	20640.0	-119.57	2.00	-124.35	-121.80	-118.49	-118.01	-
latitude	20640.0	35.63	2.14	32.54	33.93	34.26	37.71	
housing_median_age	20640.0	28.64	12.59	1.00	18.00	29.00	37.00	
total_rooms	20640.0	2635.76	2181.62	2.00	1447.75	2127.00	3148.00	39
total_bedrooms	20433.0	537.87	421.39	1.00	296.00	435.00	647.00	6
population	20640.0	1425.48	1132.46	3.00	787.00	1166.00	1725.00	35
households	20640.0	499.54	382.33	1.00	280.00	409.00	605.00	6
median_income	20640.0	3.87	1.90	0.50	2.56	3.53	4.74	
median_house_value	20640.0	206855.82	115395.62	14999.00	119600.00	179700.00	264725.00	500

5. **preprocessing** Data

5.1. **Preprocessing the data for Machine Learning algorithms**

- Shuffling the DataFrame

```
# Shuffling the DataFrame
df = shuffle(df, random_state=42)
```

- Separating the features and labels (X,y)

```
# Separating the dataset
X = df.drop("median_house_value", axis=1)
y = df["median_house_value"].copy()
```

- Splitting the dataset into training and testing sets

```
# Splitting the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_s
```

5.2. **Feature Engineering**

Creating new features that could potentially improve model performance:

- **Rooms per Household**: Total rooms in a district divided by the number of households.
- **Bedrooms per Room**: Total number of bedrooms in a district divided by the total number of rooms.
- **Population per Household**: Total population in a district divided by the number of households.

```
def add_new_features(data):
    data["rooms_per_household"] = data["total_rooms"] / data["households"]
    data["bedrooms_per_room"] = data["total_bedrooms"] / data["total_rooms"]
    data["population_per_household"] = data["population"] / data["households"]
    return data
```

```
# Applying the function to both training and testing sets
X_train = add_new_features(X_train.copy())
X_test = add_new_features(X_test.copy())
```

5.3. **Transformation Pipelines**

```
num_attribs = list(X_train.select_dtypes(include=[np.number]).columns)
cat_attribs = list(X_train.select_dtypes(exclude=[np.number]).columns)
```

```
# Creating a pipeline for numerical attributes
num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy="median")),  # Imputing missing values with the median
    ('std_scaler', StandardScaler()),  # Standardizing features by removing the mean and scaling t
])

# Encoding categorical data
cat_pipeline = Pipeline([
    ('onehot', OneHotEncoder()),  # Converting categorical variable into dummy/indicator variables
])

# Here a full preprocessing pipeline is created that applies the num and cat pipelines to their re
full_pipeline = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", cat_pipeline, cat_attribs),
])
```

6. **Modeling**

6.1. Defining the **Models for Evaluation**

- Selecting a variety of models

Goal is to cover a broad spectrum of machine learning algorithms. Here we include a mix of simple line tree-based-, random forest-, gradient boosting- and SVR-models.

```
# Defining and training multiple models
models = [
    ('Linear Regression', LinearRegression()),
    ('Decision Tree', DecisionTreeRegressor(random_state=42)),
    ('Random Forest', RandomForestRegressor(random_state=42)),
    ('Gradient Boosting', GradientBoostingRegressor(random_state=42)),
    ('Support Vector Machine', SVR())
]
```

6.2. training and testing set

[ ] ↳ *3 cells hidden*

6.3. Training and Evaluating Models Using Cross-Validation

- Training each model and evaluating its performance using cross-validation to get an estimate of its generalization error.

```
# Define the scoring metrics you want to use
scoring_metrics = ['neg_mean_squared_error', 'r2', 'neg_mean_absolute_error']
```

```
for name, model in models:
    scores = cross_validate(model,
                            X_train_prepared,
                            y_train,
                            scoring=scoring_metrics,
                            cv=10,
                            return_train_score=True)
     # Calculate the RMSE scores from neg_mean_squared_error
    rmse_scores = np.sqrt(-scores['test_neg_mean_squared_error'])

    # Access R² and MAE scores
    r2_scores = scores['test_r2']
    mae_scores = -scores['test_neg_mean_absolute_error']  # Negate to make positi

    # Print the results for each model
    print(f"{name}:")
    print(f"  Average RMSE: {np.mean(rmse_scores):.2f} ± {np.std(rmse_scores):.2f
    print(f"  Average R²: {np.mean(r2_scores):.2f} ± {np.std(r2_scores):.2f}")
    print(f"  Average MAE: {np.mean(mae_scores):.2f} ± {np.std(mae_scores):.2f}\r
```

```
Linear Regression:
  Average RMSE: 68305.59 ± 1751.25
  Average R²: 0.65 ± 0.02
  Average MAE: 49130.80 ± 757.31

Decision Tree:
  Average RMSE: 70837.87 ± 2558.68
  Average R²: 0.62 ± 0.03
  Average MAE: 45322.63 ± 1530.36

Random Forest:
  Average RMSE: 50076.61 ± 1864.54
  Average R²: 0.81 ± 0.01
  Average MAE: 32706.22 ± 1201.10

Gradient Boosting:
  Average RMSE: 53143.27 ± 1794.10
  Average R²: 0.79 ± 0.01
  Average MAE: 36852.57 ± 1095.88

Support Vector Machine:
  Average RMSE: 118050.93 ± 2155.97
  Average R²: -0.05 ± 0.01
  Average MAE: 87915.44 ± 1875.21
```

6.4. Optimizing Model

- Model Selection

Based on the cross-validation results, we are selecting the Random Forest model with the lowest average RMSE as a candidate model for further tuning and analysing.

- Hyperparameter Tuning

Applying hyperparameter tuning to models.

```
param_distributions = {
    'n_estimators': randint(100, 500),  # Assigning the number of trees in the forest
    'max_features': ['auto', 'sqrt'],  # Assigning the number of features to consider at every spl
    'max_depth': randint(10, 100),  # Assigning the maximum number of levels in tree
    'min_samples_split': randint(2, 10),  # Assigning the minimum number of samples required to sp
    'min_samples_leaf': randint(1, 4),  # Assigning the minimum number of samples required at each
    'bootstrap': [True, False]  # Assigning the method of selecting samples for training each tree
}
```

```
# Creating a Random Forest model
rf = RandomForestRegressor(random_state=42)
```

```
# Setting up the RandomizedSearchCV instance
rnd_search = RandomizedSearchCV(rf, param_distributions=param_distributions,
                                n_iter=10,
                                cv=5,
                                scoring='neg_mean_squared_error',
                                random_state=42,
                                verbose=2,
                                n_jobs=-1)
```

```
# Fitting the RandomizedSearchCV instance
rnd_search.fit(X_train_prepared, y_train)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
```

- Evaluating the Best Model from Hyperparameter Tuning

After finding the best hyperparameters using RandomizedSearchCV, we evaluate the best model on the test set to see how much the performance has improved.

```
final_model = rnd_search.best_estimator_

final_predictions = final_model.predict(X_test_prepared)
final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)

print(f"Final RMSE on Test Set: {final_rmse}")
```

```
Final RMSE on Test Set: 46598.56660232427
```

7. **Predicting**, Feature Importance and Error Analysis

7.1. **Predicting**

- Selecting Data for Prediction

```
# Selecting a small sample from the training set
some_data = X_train.iloc[:5]
some_labels = y_train.iloc[:5]
```

- Selecting the Corresponding Labels

```
# Transforming the sample data using the full preprocessing pipeline
some_data_prepared = full_pipeline.transform(some_data)
```

- Making Predictions

```
# Making predictions with the final model
some_predictions = final_model.predict(some_data_prepared)
```

```
print("Predictions:", some_predictions)
print("Actual values:", list(some_labels))
```

```
Predictions: [284145.78485309 202359.05901722 202037.62944845 206970.09259752
 318282.61531155]
Actual values: [283700.0, 190500.0, 161800.0, 171900.0, 321200.0]
```

The percentage differences between the predictions and the actual values for each of the five instances are as follows:

- 0.16% (prediction is slightly higher than the actual value)
- 6.23% (prediction is higher than the actual value)
- 24.87% (prediction is higher than the actual value)
- 20.40% (prediction is higher than the actual value)
- 0.91% (prediction is slightly lower than the actual value)

7.2. Analyzing **feature importance**

[ ] ↳ *5 cells hidden*

7.3. **Analyzing Errors**

- Analyzing errors involves looking closer at the instances where the model performed poorly. It helps us to identify any systematic errors and understand under which conditions the model is less reliable.

```
# Making predictions
predictions = final_model.predict(X_test_prepared)
```

```
# Calculating the residuals/errors
errors = y_test - predictions
```

```
# Analyzing errors
print(errors.describe())
```

```
count      4128.000000
mean       -137.443320
std       46604.009113
min     -322772.479626
25%      -22765.846607
50%       -5218.222014
75%       15698.746363
max      352081.256776
Name: median_house_value, dtype: float64
```

```
plt.hist(errors, bins=50)
plt.xlabel("Prediction Error")
plt.ylabel("Frequency")
plt.title("Distribution of Prediction Errors")
plt.show()
```