1 - DEFINE THE PROBLEM

The goal of this project is to predict the median house value for California districts based on demographic and geographic data from the 1990 census.

2 - IMPORT REQUIRED LIBRARIES

2.1 - Base Libraries

```
[1]: import pandas as pd
     import numpy as np
     import matplotlib.pyplot as plt
     import seaborn as sns
```

2.2 - MLDL Libraries

```
[2]: from sklearn.compose import ColumnTransformer
     from sklearn.impute import SimpleImputer
     from sklearn.metrics import mean_squared_error
     from sklearn.model_selection import train_test_split
     from sklearn.pipeline import Pipeline
     from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

3 - LOAD THE DATA

```
[3]: data = pd.read_csv('../datasets/housing.csv')
```

4 - EDA (Exploratory Data Analysis) of the DATA

4.1 - Basic Overview

```
[4]: data.head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY

4.2 - Check for Missing Values

```
[5]: data.isnull().sum()
```

```
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
```

4.3 - Analyze Data Types

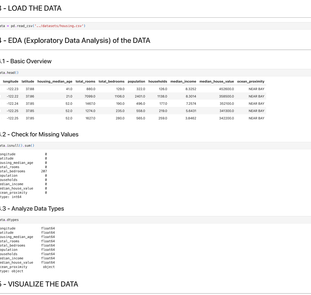
```
[6]: data.dtypes
```

```
longitude             float64
latitude              float64
housing_median_age    float64
total_rooms           float64
total_bedrooms        float64
population            float64
households            float64
median_income         float64
median_house_value    float64
ocean_proximity        object
dtype: object
```

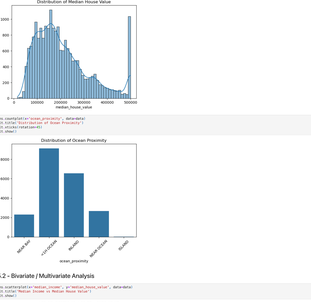
5 - VISUALIZE THE DATA

5.1 - Univariate Analysis

```
[7]: sns.histplot(data['median_house_value'], kde=True)
     plt.title('Distribution of Median House Value')
     plt.show()
```

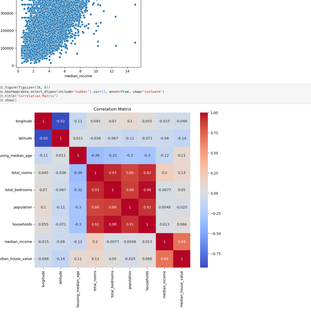



```
[8]: sns.countplot(x='ocean_proximity', data=data)
     plt.title('Distribution of Ocean Proximity')
     plt.show()
```

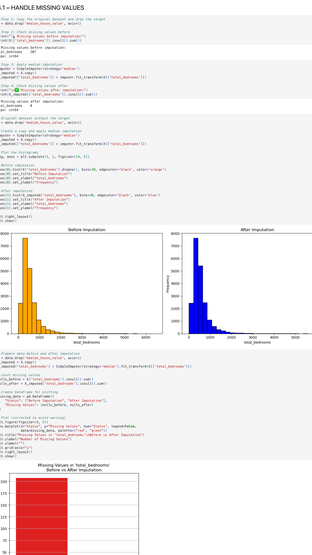


5.2 - Bivariate / Multivariate Analysis

```
[9]: sns.scatterplot(x='median_income', y='median_house_value', data=data)
     plt.title('Median Income vs Median House Value')
     plt.show()
```




```
[10]: plt.figure(figsize=(8, 8))
      sns.heatmap(data.select_dtypes(include='number').corr(), annot=True, cmap='coolwarm')
      plt.title('Correlation Matrix')
      plt.show()
```



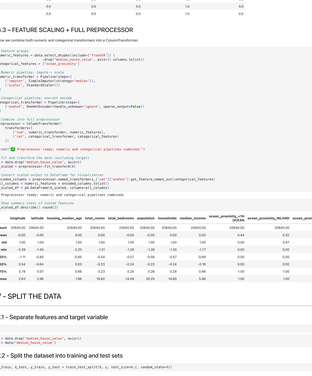
6 - PREPROCESS THE DATA

6.1 - HANDLE MISSING VALUES

```
[11]: # Step 1: Copy the original dataset and drop the target
      X = data.drop('median_house_value', axis=1)

      # Step 2: Check missing values before
      print("Missing values before imputation:")
      print(X['total_bedrooms'].isnull().sum())

      # Missing values before imputation:
      total_bedrooms    207
      dtype: int64
```

```
[12]: # Step 3: Apply median imputation
      imputer = SimpleImputer(strategy='median')
      X_imputed = X.copy()
      X_imputed['total_bedrooms'] = imputer.fit_transform(X[['total_bedrooms']])

      # Step 4: Check missing values after
      print("\nMissing values after imputation:")
      print(X_imputed['total_bedrooms'].isnull().sum())

      ✅ Missing values after imputation:
      0
```

```
[13]: # Original dataset without the target
      X = data.drop('median_house_value', axis=1)

      # Create a copy and apply median imputation
      imputer = SimpleImputer(strategy='median')
      X_imputed = X.copy()
      X_imputed['total_bedrooms'] = imputer.fit_transform(X[['total_bedrooms']])

      # Plot the histograms
      fig, axes = plt.subplots(1, 2, figsize=(14, 5))

      # Before imputation
      axes[0].hist(X['total_bedrooms'].dropna(), bins=30, edgecolor='black', color='orange')
      axes[0].set_title('Before Imputation')
      axes[0].set_xlabel('total_bedrooms')
      axes[0].set_ylabel('Frequency')

      # After imputation
      axes[1].hist(X_imputed['total_bedrooms'], bins=30, edgecolor='black', color='blue')
      axes[1].set_title('After Imputation')
      axes[1].set_xlabel('total_bedrooms')
      axes[1].set_ylabel('Frequency')

      plt.tight_layout()
      plt.show()
```




```
[14]: # Prepare data before and after imputation
      X = data.drop('median_house_value', axis=1)
      X_imputed = X.copy()
      X_imputed['total_bedrooms'] = SimpleImputer(strategy='median').fit_transform(X[['total_bedrooms']])

      # Count missing values
      nulls_before = X['total_bedrooms'].isnull().sum()
      nulls_after = X_imputed['total_bedrooms'].isnull().sum()

      # Plot
      plt.figure(figsize=(6, 4))
      missing_data = pd.DataFrame({
          'Status': ['Before Imputation', 'After Imputation'],
          'Missing Values': [nulls_before, nulls_after]
      })

      # Plot (corrected to avoid warning)
      sns.figure(figsize=(6, 4))
      sns.barplot(data=missing_data, x='Status', y='Missing Values', hue='Status', legend=False,
                  palette=['orange', 'green'])
      plt.title('Missing Values in "total_bedrooms" before vs after Imputation')
      plt.ylabel('Number of Missing Values')
      plt.xlabel('')
      plt.tight_layout()
      plt.show()
```

6.2 - ENCODE CATEGORICAL VARIABLES

```
[15]: # The column 'ocean_proximity' is categorical (object type).
      # We'll use OneHotEncoder from sklearn.preprocessing.
      categorical_features = ['ocean_proximity']

      # Step 1: Create encoder (sparse=False makes it easier to view)
      encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')

      # Step 2: Fit and transform 'ocean_proximity'
      encoded_proximity = encoder.fit_transform(data[categorical_features])

      # Step 3: Convert result to DataFrame for inspection
      encoded_df = pd.DataFrame(encoded_proximity, columns=encoder.get_feature_names_out(categorical_features))

      # Step 4: Show the result
      print("✅ One-hot encoding completed. Sample of encoded variables:")
      encoded_df.head()
```

✅ One-hot encoding completed. Sample of encoder variables:

	ocean_proximity_<1H OCEAN	ocean_proximity_INLAND	ocean_proximity_ISLAND	ocean_proximity_NEAR BAY	ocean_proximity_NEAR OCEAN
0	0.0	0.0	0.0	1.0	0.0
1	0.0	0.0	0.0	1.0	0.0
2	0.0	0.0	0.0	1.0	0.0
3	0.0	0.0	0.0	1.0	0.0
4	0.0	0.0	0.0	1.0	0.0

6.3 - FEATURE SCALING + FULL PREPROCESSOR

Now we combine both numeric and categorical transformers into a ColumnTransformer.

```
[16]: # Feature groups
      numeric_features = data.select_dtypes(include='float64').drop('median_house_value', axis=1).columns.tolist()
      categorical_features = ['ocean_proximity']

      # Numeric pipeline: impute + scale
      numeric_transformer = Pipeline(steps=[
          ('imputer', SimpleImputer(strategy='median')),
          ('scaler', StandardScaler())
      ])

      # Categorical pipeline: one-hot encode
      categorical_transformer = Pipeline(steps=[
          ('onehot', OneHotEncoder(handle_unknown='ignore', sparse_output=False))
      ])

      # Combine into full preprocessor
      preprocessor = ColumnTransformer(
          transformers=[
              ('num', numeric_transformer, numeric_features),
              ('cat', categorical_transformer, categorical_features)
          ])

      print("✅ Preprocessor ready: numeric and categorical pipelines combined.")
```

✅ Preprocessor ready: numeric and categorical pipelines combined.

```
[17]: # Fit and transform the data (excluding target)
      X = data.drop('median_house_value', axis=1)
      X_preprocessed = preprocessor.fit_transform(X)

      # Convert sparse output to dense if needed
      encoded_columns = preprocessor.named_transformers_['cat'].get_feature_names_out(categorical_features)
      X_preprocessed = pd.DataFrame(X_preprocessed, columns=numeric_features.tolist() + list(encoded_columns))

      print("✅ Preprocessor ready: numeric and categorical pipelines combined.")
```

✅ Preprocessor ready: numeric and categorical pipelines combined.

```
[18]: # Show summary stats of scaled features
      X_preprocessed.describe()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity_<1H OCEAN	ocean_proximity_INLAND	ocean_proximity_ISLAND
count	20640.00	20640.00	20640.00	20640.00	20640.00	20640.00	20640.00	20640.00	20640.00	20640.00	20640.00
mean	-0.00	-0.00	0.00	0.00	-0.00	-0.00	0.00	0.00	0.44	0.32	0.00
std	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.50	0.47	0.02
min	-2.39	-1.45	-2.20	-1.21	-1.28	-1.26	-1.30	-1.77	0.00	0.00	0.00
25%	-1.11	-0.80	-0.85	-0.54	-0.57	-0.56	-0.57	-0.69	0.00	0.00	0.00
50%	0.54	-0.64	0.03	-0.23	-0.24	-0.23	-0.24	-0.18	0.00	0.00	0.00
75%	0.78	0.97	0.66	0.23	0.25	0.26	0.26	0.46	1.00	1.00	0.00
max	2.63	2.96	1.86	18.82	14.08	30.26	14.80	5.86	1.00	1.00	1.00

7 - SPLIT THE DATA

7.1 - Separate features and target variable

```
[19]: # Define X and y
      X = data.drop('median_house_value', axis=1)
      y = data['median_house_value']
```

7.2 - Split the dataset into training and test sets

```
[20]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

7.3 - Display dimensions to verify the split

```
[21]: print("X_train:", X_train.shape)
      print("X_test:", X_test.shape)
      print("y_train:", y_train.shape)
      print("y_test:", y_test.shape)

      X_train: (16512, 9)
      X_test: (4128, 9)
      y_train: (16512,)
      y_test: (4128,)
```

```
[22]: # Step 1: Split the dataset (in case it's not already split)
      X = data.drop('median_house_value', axis=1)
      y = data['median_house_value']
      X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

      # Step 2: Compute totals and percentages
      total_rows = len(data)
      train_rows = len(X_train)
      test_rows = len(X_test)

      train_pct = round(train_rows / total_rows * 100, 2)
      test_pct = round(test_rows / total_rows * 100, 2)

      # Step 3: Define pie chart labels and sizes
      labels = [
          f'Train ({train_pct}% - {train_rows} rows)',
          f'Test ({test_pct}% - {test_rows} rows)'
      ]
      sizes = [train_rows, test_rows]
      colors = ['skyblue', 'lightgreen']

      # Step 4: Plot the pie chart
      plt.figure(figsize=(6, 6))
      plt.pie(
          sizes,
          labels=labels,
          colors=colors,
          autopct='%1.1f%%',
          startangle=140,
          wedgeprops={'edgecolor': 'black'}
      )

      plt.title(f'Train/Test Split - Total Samples: {total_rows}')
      plt.axis('equal')  # Makes the pie chart as a circle
      plt.show()
```