CHAPTER 2 END-TO-END MACHINE LEARNING PROJECT

We chose the California Housing Prices dataset from the StatLib repository.

STEP1: Looking at the Big Picture:

The first task you are asked to perform is to build a model of housing prices in California using the California census data. This data has metrics such as the population, median income, median housing price, and so on for each block group in California.

Your model should learn from this data and be able to predict the median housing price in any district, given all the other metrics.

STEP 1.1: Frame the Problem:

The first step is to clarify the business objective: understanding how the model's predictions will be used and the expected benefits. In this case, the model predicts a district's median housing price and its output feeds into a downstream Machine Learning system. This system evaluates investment opportunities in different areas, directly impacting revenue. Accurately framing the problem, selecting suitable algorithms, and defining performance measures are critical to ensuring the model aligns with business goals.

Currently, experts estimate district housing prices manually using complex rules, which are costly, slow, and have a 15% error rate. The goal is to automate this process. The problem is a **supervised learning task** since we have labeled data (housing prices) and is specifically a **regression task** because the output is a continuous value (median housing price).

STEP 1.2: Select a Performance Measure:

Your next step is to select a performance measure. A typical performance measure for regression problems is the Root Mean Square Error (RMSE), which measures the standard deviation4 of the system's errors in its predictions.

Even though the RMSE is generally the preferred performance measure for regression tasks, in some contexts, you may prefer to use another function. For example, suppose that there are many outlier districts. In that case, you may consider using the Mean Absolute Error also called the Average Absolute Deviation.

STEP 1.3 Check Assumptions:

It's important to verify assumptions early to avoid problems later. For example, if the downstream system uses price categories like "cheap," "medium," or "expensive" instead of exact prices, the task should be framed as classification, not regression. After checking with the downstream team, you confirm they need exact prices. With this clarity, you're ready to start building the system!

STEP 2: Get the data:

STEP 2.1:Create the workspace:

Create a Jupyter notebook to work on the program.

STEP 2.2: Download the Data:

In this project, you'll download a compressed file, housing.tgz, containing a housing.csv dataset. Instead of manually downloading and extracting it, writing a small function to automate this process is better. This is especially helpful if the data updates regularly, allowing you to fetch the latest version easily or schedule automated updates. It’s also practical for installing the dataset on multiple machines.

STEP 2.3: Take a quick look at the data structure:

* Let’s look at the top five rows using the DataFrame’s head() method.
* The info() method is useful to get a quick description of the data, in particular the total number of rows, and each attribute’s type and number of non-null values.
* Use the value\_counts() method to find the categories and the number of districts in each category.
* The describe() method shows a summary of the numerical attributes.

1. **Percentile**: It shows the value below which a certain percentage of data points fall.
2. **Example**: Let's say you're looking at the **housing\_median\_age** for different districts in a dataset, and you have the following percentiles:
   * **25% percentile** (1st quartile): This means 25% of the districts have a **housing\_median\_age** less than **18**.
   * **50% percentile** (Median): This is the middle value, so 50% of the districts have a **housing\_median\_age** less than **29**.
   * **75% percentile** (3rd quartile): This means 75% of the districts have a **housing\_median\_age** less than **37**.

In simpler terms:

* 25% of the data is below 18.
* 50% of the data is below 29 (this is the "middle" of the data).
* 75% of the data is below 37.

These percentiles help you understand how the data is spread out. The 25th and 75th percentiles are the "edges" of the middle 50% of the data, and the 50th percentile is the middle point of the entire dataset.

STEP 2.4: Create a Test Set:

It may seem strange to set aside part of the data early, but doing so helps avoid overfitting. If you use the test set too soon, you might find patterns that lead to a biased model selection, resulting in overly optimistic performance estimates. This is called data snooping bias. To create a test set, simply randomly select about 20% of the data and set it aside.

Random sampling can introduce bias if the dataset is small. To avoid this, **stratified sampling** is used, where the population is divided into subgroups (strata) to ensure the sample is representative. For example, a survey might ensure the sample mirrors the gender ratio of the US population. If median income is important for predicting housing prices, stratified sampling can be used to ensure the test set includes various income categories. This ensures the test set accurately reflects the full dataset.

STEP 3: Discover and visualize the data to gain insights:

STEP 3.1: VISUALIZING GEOGRAPHICAL DATA:

**Initial Visualization**: Plot districts as points on a scatterplot using their latitude and longitude. It initially shows the general shape of California, but patterns are hard to distinguish.

**Improved Density Visualization**: Setting the alpha option to 0.1 highlights high-density areas like the Bay Area, Los Angeles, San Diego, and the Central Valley (around Sacramento and Fresno).

**Advanced Visualization (Housing Prices)**:

* **Circle Radius (s)**: Represents district population.
* **Color (c)**: Represents housing prices, with a color map (cmap) that transitions from blue (low prices) to red (high prices).

This visualization shows that housing prices are influenced by location (e.g., proximity to the ocean) and population density.

**Insights**:

* Coastal areas generally have higher prices, though Northern California doesn't follow this strictly.
* Clustering algorithms could identify key patterns or clusters, adding features like proximity to cluster centers for better analysis. Ocean proximity might also be a helpful factor to consider.

STEP 3.2: LOOKING FOR COORELATION:

**Correlation** measures the strength and direction of the relationship between two variables.

* **Positive correlation**: When one variable increases, the other also increases (e.g., income and house prices).
* **Negative correlation**: When one variable increases, the other decreases (e.g., latitude and house prices in California).
* **No correlation**: No clear relationship between the variables.

 **Correlation**: Use .corr() to check how attributes relate to house prices. For example, higher **median income** strongly increases house prices.

 **Linear Relationships**: Correlation values range from -1 (negative) to 1 (positive). Near 0 means no direct link.

 **Scatterplots**: Use scatter plots to visualize these relationships. **Median income** shows the strongest link to house prices.

 **Data Quirks**: Scatterplots reveal price caps (e.g., at $500,000). Removing these quirks can improve model predictions.

STEP 3.3: EXPERIMENTING WITH ATTRIBUTE COMBINATIONS:

1. **Exploring Data**: Look for quirks, correlations, and unusual patterns in the data. Fix issues and understand key relationships.
2. **Transform Attributes**: Some attributes with uneven distributions can be transformed (e.g., using logarithms) for better analysis.
3. **New Attributes**: Create combinations for better insights:
   * **Bedrooms per room**: More correlated with house prices than total rooms or bedrooms.
   * **Rooms per household**: Larger houses are more expensive.
   * **Population per household**: Another useful metric.
4. **Iterative Process**: Start with insights and a prototype, then refine your analysis and model as you learn more.

**Summary**: Clean the data, create useful attribute combinations, and refine insights to improve predictions.

STEP:4 PREPARE THE DATA FOR ML ALGORITHM:

When preparing data for Machine Learning:

1. **Write Functions**: Automate data transformations to:
   * Reuse them on new datasets.
   * Build a reusable library for future projects.
   * Use them in live systems for consistent preprocessing.
   * Test different transformations easily.
2. **Clean Training Set**: Start with a fresh copy of the training data.
3. **Separate Predictors and Labels**: Keep input data (predictors) and output data (labels) separate, as they may need different transformations.

**Summary**: Automate data prep with functions for consistency and flexibility, and separate predictors from labels for proper processing.

STEP 4.1 DATA CLEANING:

**Handling Missing Data**:

1. **Options for Missing Values**:
   * Remove rows with missing values (dropna()).
   * Remove the entire column (drop()).
   * Replace missing values with a specific value like the median (fillna()).
2. **Best Practice**:
   * Use the median from the training set to fill missing values.
   * Save the median for future use with the test set and new data.
3. **Using Scikit-Learn's Imputer**:
   * Use SimpleImputer with strategy="median".
   * Fit the imputer to numerical data (fit()), and it calculates the median for each column.
   * Transform the data to replace missing values (transform()).
4. **Final Step**:
   * The result is a NumPy array. You can convert it back to a Pandas DataFrame for easier handling.

This ensures missing values are handled systematically and consistently.

STEP4.2: HANDLING TEXT AND CATEGORICAL ATTRIBUTES:

1. **Convert Text to Numbers**:
   * Use LabelEncoder to turn text labels into numbers (e.g., "INLAND" → 1).
   * This works for ML but assumes numeric similarity, which may not be true.
2. **Fix Numeric Assumption**:
   * Use **One-Hot Encoding** to create a binary column for each category. Only one column is "hot" (1), others are "cold" (0).
   * Scikit-Learn's OneHotEncoder does this and returns a **sparse matrix** to save memory.
3. **Combine Steps**:
   * Use LabelBinarizer for direct conversion from text to one-hot vectors.

**Summary**: Convert text categories to numerical data using one-hot encoding for better ML compatibility.

STEP 4.3: CUSTOM TRANSFORMERS:

1. **Purpose**: Create custom transformers for tasks like data cleanup or adding new features.
2. **Steps**:
   * Create a class and implement three methods: fit(), transform(), and optionally fit\_transform() (inherits from TransformerMixin for this).
   * For custom hyperparameters (like adding specific features), use an \_\_init\_\_() method to set default values.
   * fit() is for learning (usually nothing needed in custom transformers), and transform() is for changing the data.
3. **Example**:
   * A class that adds combined attributes (like rooms per household or population per household).
   * Hyperparameter add\_bedrooms\_per\_room is used to decide whether to add a new feature.

This approach allows you to automate data transformations and try multiple combinations easily.

STEP 4.4: FEATURE SCALING:

* **Why it's important**: Some Machine Learning algorithms don't work well when features have different scales (e.g., rooms count vs. income). Scaling helps make sure features are comparable.

**Two common scaling methods**:

1. **Min-Max Scaling (Normalization)**: Rescales values to a range between 0 and 1. Formula: (value - min) / (max - min).
2. **Standardization**: Subtracts the mean and divides by the standard deviation, making the data have a mean of 0 and variance of 1.

**Important**: Always fit the scaler to the **training data** only, then use it to transform both the training and test data

STEP4.5: TRANSFORMATION PIPELINES:

* A pipeline lets you apply a series of steps to your data in order, like cleaning, adding features, and scaling.
* It helps automate the data preparation process.

1. **Pipeline**:
   * First, fix missing data.
   * Then, create new features.
   * Finally, scale the data.
2. **FeatureUnion**:
   * Combines different pipelines for different types of data (e.g., numbers and text).
3. **Custom Selector**:
   * A custom step that selects the right columns of data to work with before applying transformations.

STEP 5: SELECT AND TRAIN A MODEL:

STEP 5.1: TRAINING AND EVALUATING ON THE TRAINING SET:

 **Train a Model**:

* You first train a **Linear Regression** model on your prepared data and test it by predicting values for a few samples.
* Evaluate its accuracy using **RMSE** (Root Mean Squared Error). If the error is high, it may be underfitting (not learning enough).

 **Try a More Complex Model**:

* A **Decision Tree Regressor** is more powerful and can capture complex patterns.
* It may give a perfect result on training data (zero error), but this could mean it’s **overfitting** (memorizing the data too well).

 **Validation**:

* To avoid overfitting or underfitting, use a part of the training data for validation (checking performance) instead of testing the model directly on the test set.

STEP 5.2: BETTER EVALUATION USING CROSS VALIDATION:

1. **Cross-Validation**: Instead of just splitting the data once, cross-validation splits the training data into multiple parts (e.g., 10 folds), trains the model on some, and tests on the others. This gives a better estimate of model performance.
2. **Model Comparison**:
   * For **Decision Tree**, cross-validation showed higher RMSE (around 71,200), indicating it was overfitting.
   * **Linear Regression** had a better RMSE (around 68,900), showing better generalization.
3. **Random Forest**: This model performed well with an RMSE of around 52,600, but it still had some overfitting issues.
4. **Save Models**: It's important to save your models and their settings (using joblib) to easily compare different models and return to them later.

In short, cross-validation helps to evaluate models more reliably and compare them more effectively.

STEP 6: FINE TUNE YOR MODEL:

STEP 6.1: GRID SEARCH:

1. **GridSearchCV**: This tool automates the process of searching for the best combination of hyperparameters for a model. You provide a range of values for hyperparameters, and it evaluates all possible combinations using cross-validation.
2. **Example**: For a Random Forest model, you specify values like the number of trees (n\_estimators) and features (max\_features). GridSearchCV tests all combinations (e.g., 3 values for n\_estimators and 4 for max\_features), training each combination multiple times using cross-validation.
3. **Process**: After testing, GridSearchCV will give you the best set of hyperparameters and the best model. In this example, it found the best combination as max\_features = 6 and n\_estimators = 30, with a lower RMSE than before (49,959 vs. 52,634).
4. **Additional Tuning**: You can also use GridSearchCV to fine-tune data preparation steps like adding features or handling missing values.

In short, Grid Search saves time and helps you find the best model by automating the hyperparameter tuning process.

STEP 6.2: RANDOMIZED SEARCH:

* **What it does**: Instead of testing every possible combination of hyperparameters like GridSearchCV, RandomizedSearchCV picks random combinations and evaluates them.
* **Benefits**:
  1. **More exploration**: It can test more combinations in less time (e.g., 1,000 random combinations).
  2. **Control over time**: You can set how many iterations it should run, making it easier to control the time spent on hyperparameter tuning.

In short, RandomizedSearchCV is faster and more flexible, especially when you have many hyperparameters to test.

STEP 6.3: ENSEMBLE METHODS:

Another way to fine-tune your system is to try to combine the models that perform best. The group (or “ensemble”) will often perform better than the best individual model (just like Random Forests perform better than the individual Decision Trees they rely on), especially if the individual models make very different types of errors.

STEP 6.4 ANALYSE THE BEST MODELS AND THEIR EOORS:

 **Feature Importance**: RandomForestRegressor can tell you which features (attributes) are most important for making predictions. For example, "median\_income" is the most important feature for the model, and others like "longitude" or "latitude" are less important.

 **Improvement Ideas**: Based on feature importance, you might want to remove less useful features (e.g., certain ocean proximity categories), or add new ones to improve the model.

 **Error Analysis**: It's important to look at where the model makes mistakes, understand why, and figure out how to fix them (e.g., adding or removing features, handling outliers)

STEP 6.5 EVALUATE YOUR SYSTEM ON THE TEST SET:

 **Evaluate on Test Set**: Use your full pipeline to prepare the test data (but don’t fit it again). Then, use the best model (from grid search) to make predictions on the test set and calculate the RMSE (Root Mean Squared Error).

 **Expect Slightly Worse Performance**: After tuning the model, it might perform a bit worse on the test set compared to cross-validation results. This is normal because the model was fine-tuned for the validation set and may not generalize perfectly.

 **Finalizing**: Document your findings, the strengths and weaknesses of your model, and create a clear presentation explaining key insights like the importance of certain features.

STEP 7: LAUNCH,MONITOR AND MAINTAIN YOUR SYSTEM:

After getting approval to launch, here are the steps to prepare your model for production:

1. **Connect Data Sources**: Plug your system into the real production data and set up tests to ensure everything works.
2. **Monitor Performance**: Write code to track how well the system is performing over time and set up alerts for issues like performance drops or failures.
3. **Human Evaluation**: Periodically sample and evaluate the system's predictions by using experts or crowdsourcing platforms to review them.
4. **Check Data Quality**: Monitor the input data for issues like malfunctioning sensors, which can affect model performance.
5. **Regular Retraining**: Automate the process to retrain the model with fresh data regularly. This helps prevent performance degradation and ensures the model stays up-to-date.