## **i. Machine Learning Project Lifecycle (Step-by-Step Explanation)**

A machine learning project follows a clear process from start to finish. Each step builds on the previous to ensure the model is accurate, trustworthy, and useful in real-world applications.

**1. Define the Objective**

* Understand the business or real-world problem.
* Determine if the task is:
  + **Regression** (predict numbers)
  + **Classification** (predict categories)
  + **Clustering** (group similar items)
* Example: Predict median house prices in California based on various housing features.

**2. Get the Data**

* Collect data from:
  + Public datasets
  + APIs
  + Company databases
* In this project, we use the **California Housing Dataset**, which contains information like population, median income, and number of rooms.

python

CopyEdit

from sklearn.datasets import fetch\_california\_housing

housing = fetch\_california\_housing(as\_frame=True).frame

**3. Explore the Data**

* Check data structure:
  + Use .head(), .info(), .describe() to understand the columns, types, and summary stats.
* Use **histograms** and **correlation matrices** to detect patterns, skewed features, or outliers.

**4. Create a Test Set**

* Before deep analysis, split data into:
  + **Training set** (e.g., 80%)
  + **Test set** (e.g., 20%)
* This prevents **data leakage** and ensures realistic evaluation later.

python

CopyEdit

from sklearn.model\_selection import train\_test\_split

train\_set, test\_set = train\_test\_split(housing, test\_size=0.2, random\_state=42)

**5. Data Preprocessing**

* Handle **missing values**:
  + Drop rows or fill (imputation using median, mean, or most frequent).
* Deal with **text or categorical data** using:
  + Label encoding
  + One-hot encoding
* Scale features (e.g., StandardScaler) to bring all to a similar range.

**6. Feature Engineering**

* Create new features that make patterns easier to learn.
* Example from the dataset:
  + rooms\_per\_household = total\_rooms / households
  + bedrooms\_per\_room = total\_bedrooms / total\_rooms
  + population\_per\_household = population / households

These can make models more accurate.

**7. Model Training**

* Train models like:
  + **Linear Regression**
  + **Decision Trees**
  + **Random Forest**
* Evaluate using metrics like:
  + **Mean Squared Error (MSE)**
  + **Root Mean Squared Error (RMSE)**

**8. Model Fine-Tuning**

* Improve performance by adjusting hyperparameters:
  + Use **Grid Search** or **Randomized Search** to try multiple combinations.
  + Apply **cross-validation** for reliable scores.

python

CopyEdit

from sklearn.model\_selection import GridSearchCV

# Example code for GridSearch

**9. Evaluate Final Model**

* Evaluate the best model on the untouched **test set**.
* Report final RMSE or accuracy.
* This gives the best estimate of real-world performance.

**10. Deploy the Model**

* Save the trained model using joblib or pickle.
* Deploy using:
  + **Flask**, **FastAPI** (Python web frameworks)
  + **TensorFlow Serving**
  + **Mobile apps** using **TensorFlow Lite**

**ii. Stratified Sampling Concept (In Detail)**

**What is the Problem with Simple Random Sampling?**

* Random sampling might not maintain the same distribution as the original data.
* Example: If high-income people are rare, they might get underrepresented.

**What is Stratified Sampling?**

* Ensures the **train and test sets** contain the same **proportions** of important subgroups.
* A **stratum** is a category based on an important feature (like income category).
* Helps to:
  + Reduce **sampling bias**
  + Increase **model performance reliability**

**Example (With Income Categories):**

We categorize incomes into **5 levels**:

python

CopyEdit

housing["income\_cat"] = pd.cut(housing["median\_income"],

bins=[0., 1.5, 3.0, 4.5, 6., np.inf],

labels=[1, 2, 3, 4, 5])

Then apply stratified sampling:

python

CopyEdit

from sklearn.model\_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n\_splits=1, test\_size=0.2, random\_state=42)

for train\_idx, test\_idx in split.split(housing, housing["income\_cat"]):

strat\_train\_set = housing.loc[train\_idx]

strat\_test\_set = housing.loc[test\_idx]

This way, both sets maintain the same **income category distribution**.

**iii. Feature Engineering Pipeline (Detailed Explanation)**

Instead of manually transforming data, use **pipelines** to make preprocessing:

* **Automatic**
* **Reusable**
* **Clean and modular**

**Step-by-Step Construction of a Pipeline**

**1. Numerical Pipeline:**

* Impute missing values (e.g., with median)
* Scale the features (e.g., using StandardScaler)

python

CopyEdit

from sklearn.pipeline import Pipeline

from sklearn.impute import SimpleImputer

from sklearn.preprocessing import StandardScaler

num\_pipeline = Pipeline([

('imputer', SimpleImputer(strategy='median')),

('scaler', StandardScaler()),

])

**2. Handle Categorical Columns:**

* Convert categories into binary columns (one-hot encoding)

python

CopyEdit

from sklearn.preprocessing import OneHotEncoder

**3. Combine Using ColumnTransformer:**

* Apply different transformations to different types of columns.

python

CopyEdit

from sklearn.compose import ColumnTransformer

full\_pipeline = ColumnTransformer([

('num', num\_pipeline, num\_attribs),

('cat', OneHotEncoder(), cat\_attribs),

])

Now you can transform the data with:

python

CopyEdit

housing\_prepared = full\_pipeline.fit\_transform(housing)

**Benefits of Feature Pipelines:**

* Ensures consistent preprocessing during:
  + Training
  + Evaluation
  + Production deployment
* Reduces errors
* Keeps code **organized and modular**

**ML Project Lifecycle (Chapter 2)**

**From the book: *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow – Chapter 2***

**What Is the ML Project Lifecycle?**

The **Machine Learning (ML) Project Lifecycle** is a systematic process that guides you from the start (problem definition) to the final product (deployment).  
This ensures your model is:

* Built using clean, relevant data
* Trained and tested properly
* Delivered as a useful, real-world application

**Detailed Lifecycle Steps (As Applied in Your Project)**

|  |  |  |
| --- | --- | --- |
| **🔢 Step** | **🔍 Description** | **💡 What You Did in Project** |
| **1. Define the Problem** | Identify if it’s regression or classification. Choose the target. | You framed this as a **regression** problem to predict MedHouseVal (Median House Value). |
| **2. Acquire the Data** | Collect data from files, APIs, or datasets. | Used the built-in **California Housing dataset** from sklearn.datasets.fetch\_california\_housing(). |
| **3. Explore the Data** | Use pandas, matplotlib, and seaborn to understand the structure, stats, and patterns. | You plotted histograms, correlation heatmaps, scatter plots, and a housing map. |
| **4. Prepare the Data** | Handle missing values, scale features, and engineer new ones. | Used SimpleImputer, StandardScaler, and created new features like rooms\_per\_household. |
| **5. Select a Model** | Try multiple models that fit your problem. | Trained **Linear Regression**, **Decision Tree**, **Random Forest**, and **SVR**. |
| **6. Train the Model** | Fit the model to training data. | Used .fit() on X\_train\_prepared and y\_train. |
| **7. Evaluate the Model** | Test model accuracy with RMSE and MAE. Compare models. | Measured performance using mean\_squared\_error() and mean\_absolute\_error(). |
| **8. Tune Hyperparameters** | Improve model by optimizing parameters. | Used **RandomizedSearchCV** and **GridSearchCV** (SVR tuning example). |
| **9. Deploy the Model** | Build a system/app to use the trained model. | Built a **Streamlit app** with sliders and predictions shown on screen. |
| **10. Maintain and Monitor** | Keep improving based on real-world use. | (Optional in this project, but important for real systems.) |

**Key Concept Explained: “What Is a Regression Problem?”**

A **regression** problem is when the **target/output** is a **continuous numerical value**.  
In your case:

* Target: MedHouseVal → Example value: 2.342  
  This makes it a **supervised regression task**, not classification.

**🧰 Tools Used at Each Step**

|  |  |
| --- | --- |
| **Phase** | **Tools/Library** |
| Data loading | fetch\_california\_housing() |
| Exploration | pandas, matplotlib, seaborn |
| Preprocessing | SimpleImputer, StandardScaler, Pipeline |
| Modeling | sklearn.linear\_model, sklearn.tree, sklearn.ensemble, sklearn.svm |
| Evaluation | mean\_squared\_error, mean\_absolute\_error |
| Deployment | streamlit |

**Why This Lifecycle Is Important:**

1. **Avoids mistakes** like data leakage or overfitting.
2. **Makes your project reproducible** and easy to understand.
3. **Standard process used in real industry ML teams.**

## What is Stratified Sampling?

**Stratified Sampling** is a data sampling technique used to ensure that each **important subgroup (stratum)** in the dataset is represented proportionally in both the **training** and **test sets**.

💡 Instead of selecting random samples, stratified sampling guarantees that the **distribution of a specific feature** (e.g., income level) stays the same across splits.

## Why It Matters in ML?

If the **test set is not representative**, your model might:

* Perform well on test data but poorly in real-world deployment.
* Miss rare but important categories (like low-income districts).

**Random sampling** can accidentally skip some important categories.

## Example from California Housing Project

### Goal:

Ensure that **income categories** are distributed fairly in both training and test sets.

### How It Was Done:

python

CopyEdit

housing["income\_cat"] = pd.cut(

housing["MedInc"],

bins=[0., 1.5, 3.0, 4.5, 6., np.inf],

labels=[1, 2, 3, 4, 5]

)

This created **5 income categories** based on MedInc (Median Income), turning it into a **categorical feature**.

Then used **StratifiedShuffleSplit**:

python

CopyEdit

from sklearn.model\_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n\_splits=1, test\_size=0.2, random\_state=42)

for train\_idx, test\_idx in split.split(housing, housing["income\_cat"]):

strat\_train\_set = housing.loc[train\_idx]

strat\_test\_set = housing.loc[test\_idx]

## Why We Use StratifiedShuffleSplit Instead of train\_test\_split

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Is it random?** | **Maintains distribution?** | **Suitable for category balance?** |
| train\_test\_split() | Yes | No | Risk of skewed data |
| StratifiedShuffleSplit() | Yes with control | Yes | Ideal for classification or balanced regression |

## Visual Check (Optional)

You can plot a histogram of the income\_cat column in full, train, and test sets to confirm that the proportions match.

python

CopyEdit

housing["income\_cat"].value\_counts() / len(housing)

strat\_train\_set["income\_cat"].value\_counts() / len(strat\_train\_set)

strat\_test\_set["income\_cat"].value\_counts() / len(strat\_test\_set)

All will be very similar if stratified sampling worked properly.

## 🧩 Key Takeaways

|  |  |
| --- | --- |
| **Benefit** | **Explanation** |
| Better evaluation | Prevents misleading accuracy from biased test sets. |
| Real-world representativeness | Mirrors the actual population structure. |
| More consistent performance results | Stable metrics across reruns. |

**Feature Engineering Pipeline**

**What is Feature Engineering?**

**Feature Engineering** is the process of:

Creating new input features or modifying existing ones to help the model learn patterns more effectively.

**Why Is It Important?**

Even if you have lots of data, your model can **fail to learn properly** if your features are:

* Redundant
* Unscaled
* Missing key relationships

**What Did You Do in the Project?**

You used **feature engineering** to:

|  |  |  |
| --- | --- | --- |
| **New Feature** | **Formula** | **Purpose** |
| rooms\_per\_household | AveRooms / AveOccup | Normalizes room count per occupied home |
| bedrooms\_per\_room | AveBedrms / AveRooms | Helps detect overcrowding |
| income\_per\_room | MedInc / AveRooms | Captures income relative to space |

These features captured **non-obvious relationships** that boosted model performance.

**What Is a Pipeline?**

A **pipeline** in scikit-learn lets you chain **data transformation steps** (like imputation, scaling, and feature addition) together into a single object.

It ensures **consistency** and **avoids data leakage**, especially between training and testing sets.

**Basic Pipeline You Used**

python

CopyEdit

from sklearn.pipeline import Pipeline

from sklearn.impute import SimpleImputer

from sklearn.preprocessing import StandardScaler

num\_pipeline = Pipeline([

("imputer", SimpleImputer(strategy="median")),

("scaler", StandardScaler())

])

* **Step 1:** Fill missing values (SimpleImputer)
* **Step 2:** Normalize features (StandardScaler)

Then:

python

CopyEdit

X\_train\_prepared = num\_pipeline.fit\_transform(X\_train)

X\_test\_prepared = num\_pipeline.transform(X\_test)

You use fit\_transform() only on training data  
Use transform() on test data to avoid leaking test information into the training process

**Full Data Prep + Modeling Pipeline (Advanced)**

Later in the exercises, you wrapped **preprocessing + model** in a single pipeline:

python

CopyEdit

full\_pipeline = Pipeline([

("preprocessing", num\_pipeline),

("model", RandomForestRegressor())

])

Now you can do:

python

CopyEdit

full\_pipeline.fit(X\_train, y\_train)

predictions = full\_pipeline.predict(X\_test)

This makes:

* Model deployment much easier
* Hyperparameter search more modular
* Everything clean and reproducible

**Summary Table: Feature Engineering Concepts**

|  |  |
| --- | --- |
| **Term** | **Explanation** |
| SimpleImputer | Fills in missing values (median, mean, etc.) |
| StandardScaler | Scales numerical features (mean=0, std=1) |
| Pipeline | Chains multiple steps together |
| fit\_transform() | Trains + transforms data (used only on training set) |
| transform() | Applies the same transformation (used on test set) |
| Engineered Features | Derived attributes like rooms\_per\_household |