Machine Learning Problem-Solving Steps

In order to solve machine learning problems, we follow a structured approach that helps ensure accuracy, clarity, and effectiveness. Here are the main steps involved:

1. Look at the Big Picture

Understand the overall problem you're solving. Define your objective clearly — what does success look like?

2. Get the Data

Collect relevant and quality data from reliable sources. Without data, there's no machine learning.

3. Explore and Visualize the Data

Analyze and visualize data to uncover patterns, trends, and anomalies. This step helps you understand what you're working with.

4. Prepare the Data

Clean, transform, and format the data. Handle missing values, normalize features, and split the data into training and testing sets.

5. Select a Model and Train It

Choose a suitable machine learning algorithm and train it using your data. This is where your model learns from patterns.

6. Fine-Tune Your Model

Optimize hyperparameters, try different techniques, and improve performance through iteration.

7. Present Your Solution

Explain your model's results using visuals, metrics, and clear language so stakeholders can understand and make decisions.

8. Launch, Monitor, and Maintain

Deploy the model in the real world, monitor its performance, and update it regularly as new data arrives.

Datasets for Machine Learning

Machine Learning requires quality datasets for training and testing models. Some popular sources include:

- **OpenML.org** – A collaborative platform offering a wide range of datasets with metadata and tools for benchmarking.
- **UCI Machine Learning Repository** – One of the oldest and most widely used sources for machine learning datasets.
- **Kaggle** – A data science community offering large-scale, real-world datasets and competitions.

Quick Training with Scikit-learn (CSV Data)

This guide walks you through training a model on your own CSV dataset using scikit-learn.

1. Import Libraries

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
```

2. Load Your CSV File

```
data = pd.read_csv('data.csv')  # Replace with your actual CSV file path
```

3. Separate Features and Label

```
X = data.iloc[:, :-1]  # All columns except the last as features
y = data.iloc[:, -1]   # Last column as label
```

4. Train the Model

```
model = RandomForestClassifier()
model.fit(X, y)
```

The model is now trained on your complete dataset.

5. Inference

```
predictions = model.predict(X)  # Predict on the same/new data (for demonstration)
print(predictions)  # Display predictions
```

Evaluating Performance of ML Models

When we build a machine learning model, especially a classification model (which predicts categories like “spam” or “not spam”, “dog” or “cat”), it’s important to measure how well the model is performing.

One of the most basic ways to evaluate a classification model is **accuracy**.

What is Accuracy?

In simple terms, **accuracy** tells us how often the model was **right**.

If you gave your model 100 questions to answer, and it got 90 of them correct, its accuracy would be:

```
Accuracy = Correct Predictions / Total Predictions = 90 / 100 = 90%
```

So, **accuracy** is just the fraction of predictions the model got right.

Evaluating Accuracy

> **Accuracy = (Correct Predictions) / (Total Predictions)**

This is the simplest and most intuitive way to check if the model is doing a good job.

Gurgaon House Price Prediction Model

Why Are We Building This?

Gurgaon, a rapidly growing city in India, has seen a sharp rise in real estate development over the past decade. With its proximity to Delhi, booming IT hubs, and modern infrastructure, Gurgaon has become a major attraction for both homebuyers and investors. However, the real estate market here is highly dynamic and often difficult to assess without proper data-driven tools.

We're building a Gurgaon house price prediction model to help:

- Understand how factors like location, size, number of rooms, and amenities affect property prices in Gurgaon.
- Assist buyers in identifying fair prices based on historical trends.
- Help sellers estimate an appropriate asking price.
- Empower real estate agents and platforms to improve recommendations and negotiations.

How Will We Build It?

While we don't have access to a large, clean dataset of house prices in Gurgaon right now, we will use a well-known and cleaned dataset—the California housing dataset—as a proxy. This will allow us to build, test, and evaluate a working model with real-world variables like:

- Median income of the area
- Proximity to the city center
- Number of rooms
- Latitude and longitude
- Population density

We'll treat this as a simulation: suppose the California data is Gurgaon data, and suppose we are building this model for a neighborhood where both you and I live or work nearby.

Once the model is developed and understood, we can later adapt the same approach to real Gurgaon data when available, using the same techniques and logic.

Revisiting Steps to solve this problem

Understanding the Problem

Before we build the model, we need to understand what kind of machine learning problem we are solving.

First, we'll check if this is a **supervised** or **unsupervised** learning task. Since we have historical data with house prices (our target), and we want to predict the price of a house based on input features, this is a **supervised learning** problem.

Next, we observe that the model is predicting **one continuous label** — the price of a house — based on several input features. This makes it a **univariate regression problem**.

Measuring Errors (RMSE & MAE)

After training our regression model, we need to evaluate how good its predictions are. Two common metrics used for this are **MAE** and **RMSE**.

1. Mean Absolute Error (MAE)

MAE stands for Mean Absolute Error. It calculates the average of the absolute differences between the predicted and actual values.

Formula: MAE = (1/n) × sum of |actual - predicted|

- It treats all errors equally, no matter their size.
- MAE is based on the **Manhattan norm** (also called L1 norm), which measures distance by summing absolute values.

2. Root Mean Squared Error (RMSE)

RMSE stands for Root Mean Squared Error. It calculates the square root of the average of squared differences between predicted and actual values.

Formula: RMSE = sqrt((1/n) × sum of $(\text{actual} - \text{predicted})^2$)

- RMSE gives more weight to larger errors because it squares them.
- RMSE is based on the **Euclidean norm** (also called L2 norm), which measures straight-line distance.

Summary

- **Use MAE** when all errors should be treated equally.
- **Use RMSE** when larger errors should be penalized more.

Analyzing the Data (EDA)

Exploratory Data Analysis (EDA) is the process of examining a dataset to summarize its main characteristics, often using visual methods or quick commands. The goal is to understand the structure of the data, detect patterns, spot anomalies, and get a feel for what kind of preprocessing or modeling might be needed.

For our project, we'll perform EDA on the California housing dataset (which we are treating as if it represents Gurgaon data). Here are some key commands we'll use:

1. `df.head()`

- Displays the first 5 rows of the dataset.
- Useful for getting a quick overview of what the data looks like — column names, data types, and sample values.

2. `df.info()`

- Gives a summary of the dataset.
- Shows the number of entries, column names, data types, and how many non-null values each column has.
- Helps us identify missing values or incorrect data types.

3. `df.describe()`

- Provides statistical summaries for numeric columns.
- Shows:
    - **Count**: Total number of non-null entries
    - **Mean**: Average value
    - **Std**: Standard deviation
    - **Min**: The smallest value (0th percentile or 1st quartile in some contexts)

- **25%**: The 1st quartile (Q1) — 25% of the data is below this value
- **50%**: The median or 2nd quartile (Q2) — half of the data is below this value
- **75%**: The 3rd quartile (Q3) — 75% of the data is below this value
- **Max**: The largest value (often considered the 4th quartile or 100th percentile)

> **Percentiles** divide the data into 100 equal parts. **Quartiles** divide the data into 4 equal parts (Q1 = 25th percentile, Q2 = 50th, Q3 = 75th). So:
>
> - **Min** is the 0th percentile
> - **Max** is the 100th percentile

This helps us understand how the values are spread out and if there are outliers.

4. `df['column_name'].value_counts()`

- Shows the count of each unique value in a specific column.
- Useful for categorical columns to see how values are distributed.

Creating a Test Set

When building machine learning models, one of the most important steps is **splitting your dataset** into training and test sets. This ensures your model is evaluated on data it has never seen before, which is critical for assessing its ability to generalize.

The Problem of Data Snooping Bias

Data snooping bias occurs when information from the test set leaks into the training process. This can lead to overly optimistic performance metrics and models that don't perform well in real-world scenarios.

To avoid this, the test set must be isolated before any data exploration, feature selection, or model training begins.

Random Sampling: A Basic Approach

A simple method to split the data is to randomly shuffle it and then divide it:

```
import numpy as np

def shuffle_and_split_data(data, test_ratio):
    np.random.seed(42)  # Set the seed for reproducibility
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

Setting the random seed (e.g., with `np.random.seed(42)`) ensures consistency across runs — this is crucial for debugging and comparing models fairly.

However, pure random sampling might not always be reliable, especially if the dataset contains important patterns that are not evenly distributed.

Stratified Sampling

To ensure that important characteristics of the population are well represented in both the training and test sets, we use **stratified sampling**.

What is a Strata?

A **strata** is a subgroup of the data defined by a specific attribute. Stratified sampling ensures that each of these subgroups is proportionally represented.

For example, in the California housing dataset, **median income** is a strong predictor of house prices. Instead of randomly sampling, we can create strata based on income levels (e.g., binning median income into categories) and ensure the test set maintains the same distribution of income levels as the full dataset.

Creating Income Categories

```
import pandas as pd
# Load the dataset
data = pd.read_csv("housing.csv")
# Create income categories
data["income_cat"] = pd.cut(data["median_income"],
                            bins=[0, 1.5, 3.0, 4.5, 6.0, np.inf],
                            labels=[1, 2, 3, 4, 5])
```

This code creates a new column `income_cat` that categorizes the `median_income` into five bins. Each bin represents a range of income levels, allowing us to stratify our sampling based on these categories.

We can plot these income categories to visualize the distribution:

```
import matplotlib.pyplot as plt
data["income_cat"].value_counts().sort_index().plot.bar(rot=0, grid=True)
plt.title("Income Categories Distribution")
plt.xlabel("Income Category")
```

```
plt.ylabel("Number of Instances")
plt.show()
```

Stratified Shuffle Split in Scikit-Learn

Scikit-learn provides a built-in way to perform stratified sampling using `StratifiedShuffleSplit`.

Here's how you can use it:

```
from sklearn.model_selection import StratifiedShuffleSplit

# Assume income_cat is a column in the dataset created from median_income
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)

for train_index, test_index in split.split(data, data["income_cat"]):
    strat_train_set = data.loc[train_index]
    strat_test_set = data.loc[test_index]
```

This ensures that the income distribution in both sets is similar to that of the full dataset, reducing sampling bias and making your model evaluation more reliable.

Data Visualization

Before handling missing values or training models, it's important to **visualize the data** to uncover patterns, relationships, and potential issues.

Geographical Scatter Plot

Visualize the geographical distribution of the data:

```
df.plot(kind="scatter", x="longitude", y="latitude", grid=True, alpha=0.2)
plt.show()
```

- `alpha=0.2` makes overlapping points more visible.
- This helps reveal data clusters and high-density areas like coastal regions.

Correlation Matrix

To understand relationships between numerical features, compute the **correlation matrix**:

```
corr_matrix = df.corr()
```

Check how strongly each attribute correlates with the target:

```
corr_matrix["median_house_value"].sort_values(ascending=False)
```

This helps identify useful predictors. For example, `median_income` usually shows a strong positive correlation with house prices.

Scatter Matrix

Plot selected features to see pairwise relationships:

```
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms", "housing_median
_age"]
scatter_matrix(df[attributes], figsize=(12, 8))
plt.show()
```

This gives an overview of which features are linearly related and may be good predictors.

Focused Income vs Price Plot

Plot `median_income` vs `median_house_value` directly:

```
df.plot(kind="scatter", x="median_income", y="median_house_value", alpha=0.1,
grid=True)
```

Further Preprocessing & Handling Missing Data

Before feeding your data into a machine learning algorithm, you need to clean and prepare it.

Prepare Data for Training

It's best to write transformation functions instead of applying them manually. This ensures:

- Reproducibility on any dataset
- Reusability across projects
- Compatibility with live systems
- Easier experimentation

Start by creating a clean copy and separating the predictors and labels:

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

Handling Missing Data

Some features, like `total_bedrooms`, contain missing values. You can:

1. Drop rows with missing values
2. Drop the entire column
3. **Impute missing values** (recommended)

We'll use **option 3** using `SimpleImputer` from Scikit-Learn, which allows consistent handling across all datasets (train, test, new data):

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")

housing_num = housing.select_dtypes(include=[np.number])
imputer.fit(housing_num)
```

This computes the median for each numerical column and stores it in `imputer.statistics_`:

```
>>> imputer.statistics_
array([-118.51 ,  34.26 ,  29. , 2125. , 434. , 1167. , 408. , 3.5385])
```

Now apply the learned medians to transform the data:

```
X = imputer.transform(housing_num)
```

Other available strategies:

- `"mean"` – replaces with mean value
- `"most_frequent"` – for the most common value (can handle categorical)
- `"constant"` – fill with a fixed value using `fill_value=...`

Scikit-Learn Design Principles

Scikit-Learn has a simple and consistent API that makes it easy to use and understand. Below are the key design principles behind it:

---

1. Consistency

All objects follow a standard interface, which makes learning and using different tools in Scikit-Learn easier.

---

2. Estimators

Any object that learns from data is called an **estimator**.

- Use the `.fit()` method to train an estimator.
- In supervised learning, pass both `X` (features) and `y` (labels) to `.fit(X, y)`.
- Hyperparameters (like `strategy='mean'` in `SimpleImputer`) are set when creating the object.

Example:

```
imputer = SimpleImputer(strategy="median")
imputer.fit(data)
```

---

3. Transformers

Some estimators can also transform data. These are called **transformers**.

- Use `.transform()` to apply the transformation after fitting.
- Use `.fit_transform()` to do both in one step.

Example:

```
X_transformed = imputer.fit_transform(data)
```

---

4. Predictors

Models that can make predictions are **predictors**.

- Use `.predict()` to make predictions on new data.
- Use `.score()` to evaluate performance (e.g., accuracy or R^2).

Example:

```
model = LinearRegression()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
score = model.score(X_test, y_test)
```

---

5. Inspection

- Hyperparameters can be accessed directly: `model.param_name`
- Learned parameters are stored with an underscore: `model.coef_`, `imputer.statistics_`

---

6. No Extra Classes

- Inputs and outputs are basic structures like NumPy arrays or Pandas DataFrames.
- No need to learn custom data types.

---

7. Composition

You can combine steps into a **Pipeline**, chaining transformers and a final predictor.

Example:

```
pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="median")),
    ("model", LinearRegression())
])
pipeline.fit(X, y)
```

---

8. Sensible Defaults

Most tools in Scikit-Learn work well with default settings, so you can get started quickly.

Note on DataFrames

Even if you input a Pandas DataFrame, the output of transformers like `transform()` will be a NumPy array. You can convert it back like this:

```
X = imputer.transform(housing_num)
housing_tr = pd.DataFrame(X, columns=housing_num.columns, index=housing_num.index)
```

Handling Categorical and Text Attributes in Scikit-Learn

Most machine learning algorithms work best with numerical data. But real-world datasets often contain **categorical** or **text attributes**. Let's understand how to handle these in Scikit-Learn using the `ocean_proximity` column from the California housing dataset as an example.

---

1. Categorical Attributes

Text columns like `"ocean_proximity"` are not free-form text but limited to a fixed set of values (e.g., `"NEAR BAY"`, `"INLAND"`). These are known as **categorical attributes**.

Example:

```
housing_cat = housing[["ocean_proximity"]]
housing_cat.head()
```

---

2. Ordinal Encoding

Scikit-Learn's `OrdinalEncoder` can convert categories to numbers:

```
from sklearn.preprocessing import OrdinalEncoder


ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
```

This will output a 2D NumPy array with numerical category codes.

To see the mapping:

```
ordinal_encoder.categories_
# Output: array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'])
```

⚠ **Caution**: Ordinal encoding implies an order between categories, which may not be true here. For example, it treats `INLAND (1)` as closer to `<1H OCEAN (0)` than `NEAR OCEAN (4)`, which might not make sense.

---

3. One-Hot Encoding

For unordered categories, **one-hot encoding** is a better choice. It creates one binary column per category.

```
from sklearn.preprocessing import OneHotEncoder

cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

This gives a **sparse matrix** (efficient storage for mostly zeros).

To convert it to a regular NumPy array:

```
housing_cat_1hot.toarray()
```

Or directly get a dense array:

```
cat_encoder = OneHotEncoder(sparse=False)
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

To check category order:

```
cat_encoder.categories_
```

4. Summary

Method	Use When	Output Type
`OrdinalEncoder`	Categories have an order	2D NumPy array
`OneHotEncoder`	Categories are unordered	Sparse or dense

Using the right encoding ensures your model learns correctly from categorical features.

Feature Scaling and Transformation

Feature scaling is a crucial preprocessing step. Most machine learning algorithms perform poorly when input features have vastly different scales.

In the California housing dataset, for example:

- `total_rooms` ranges from 6 to over 39,000
- `median_income` ranges from 0 to 15

If you don't scale these features, models will give more importance to `total_rooms` simply because it has larger values.

Why Scaling Is Needed

- Many models (like Linear Regression, KNN, SVMs, Gradient Descent-based algorithms) **assume features are on a similar scale**.
- Without scaling, features with larger ranges can **dominate model behavior**.
- Scaling makes training **more stable and faster**.

Min-Max Scaling (Normalization)

This method rescales the data to a specific range, usually `[0, 1]` or `[-1, 1]`.

Formula:

```
scaled_value = (x - min) / (max - min)
```

Use Scikit-Learn's `MinMaxScaler`:

```
from sklearn.preprocessing import MinMaxScaler
```

```
min_max_scaler = MinMaxScaler(feature_range=(-1, 1))
housing_num_min_max_scaled = min_max_scaler.fit_transform(housing_num)
```

- Use `feature_range=(-1, 1)` for models like neural networks.
- Sensitive to outliers — extreme values can distort the scale.

Standardization (Z-score Scaling)

This method centers the data around 0 and scales it based on standard deviation.

Formula:

```
standardized_value = (x - mean) / std
```

Use Scikit-Learn's `StandardScaler`:

```
from sklearn.preprocessing import StandardScaler

std_scaler = StandardScaler()
housing_num_std_scaled = std_scaler.fit_transform(housing_num)
```

- Resulting features have **zero mean and unit variance**
- **Robust to outliers** compared to min-max scaling
- Recommended for most ML algorithms, especially when using gradient descent

Transformation Pipelines

As datasets grow more complex, data preprocessing often involves multiple steps such as imputing missing values, scaling features, encoding categorical variables, etc. These steps must be applied **in the correct order** and consistently across training, validation, test, and future production data.

To streamline this process, **Scikit-Learn** provides the `Pipeline` class — a powerful utility for chaining data transformations.

Building a Numerical Pipeline

A typical pipeline for numerical attributes might include:

1. **Imputation** of missing values (e.g., with median).
2. **Feature scaling** (e.g., with standardization).

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
    ("impute", SimpleImputer(strategy="median")),
    ("standardize", StandardScaler()),
])
```

How It Works

- The pipeline takes a list of steps as `(name, transformer)` pairs.
- Names must be unique and should not contain double underscores `__`.
- All intermediate steps must be transformers (i.e., must implement `fit_transform()`).

- The final step can be either a transformer or a predictor.

Using `make_pipeline`

If you don't want to name the steps manually, you can use `make_pipeline()`:

```
from sklearn.pipeline import make_pipeline

num_pipeline = make_pipeline(SimpleImputer(strategy="median"), StandardScaler())
```

- This automatically names the steps using the class names in lowercase.
- If the same class appears multiple times, a number is appended (e.g., `standardscaler-1`).

Applying the Pipeline

Call `fit_transform()` to apply all transformations in sequence:

```
housing_num_prepared = num_pipeline.fit_transform(housing_num)
print(housing_num_prepared[:2].round(2))
```

Example output:

```
array([[-1.42,  1.01,  1.86,  0.31,  1.37,  0.14,  1.39, -0.94],
       [ 0.60, -0.70,  0.91, -0.31, -0.44, -0.69, -0.37,  1.17]])
```

- Each row corresponds to a transformed sample.
- Each column corresponds to a scaled feature.

Retrieving Feature Names

To turn the result back into a DataFrame with feature names:

```
df_housing_num_prepared = pd.DataFrame(
    housing_num_prepared,
    columns=num_pipeline.get_feature_names_out(),
    index=housing_num.index
)
```

Pipeline as a Transformer or Predictor

- If the **last step is a transformer**, the pipeline behaves like a transformer (`fit_transform()` , `transform()`).
- If the **last step is a predictor** (e.g., a model), the pipeline behaves like an estimator (`fit()` , `predict()`).

This flexibility makes `Pipeline` the standard way to handle data preprocessing and modeling in Scikit-Learn projects.

Data Preprocessing - Final Pipeline

In this section, we will **consolidate everything** we've done so far into one final script using Scikit-Learn pipelines. This includes:

1. Creating a stratified test set
2. Handling missing values
3. Encoding categorical variables
4. Scaling numerical features
5. Combining everything using `Pipeline` and `ColumnTransformer`

This will ensure **clean, modular, and reproducible code** — perfect for production and education.

Final Preprocessing Code using Scikit-Learn Pipelines

```
import pandas as pd
import numpy as np
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
# from sklearn.preprocessing import OrdinalEncoder  # Uncomment if you prefer
ordinal

# 1. Load the data
housing = pd.read_csv("housing.csv")

# 2. Create a stratified test set based on income category
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]
)

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index].drop("income_cat", axis=1)
    strat_test_set = housing.loc[test_index].drop("income_cat", axis=1)

# Work on a copy of training data
housing = strat_train_set.copy()

# 3. Separate predictors and labels
housing_labels = housing["median_house_value"].copy()
housing = housing.drop("median_house_value", axis=1)

# 4. Separate numerical and categorical columns
num_attribs = housing.drop("ocean_proximity", axis=1).columns.tolist()
cat_attribs = ["ocean_proximity"]

# 5. Pipelines
# Numerical pipeline
num_pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler", StandardScaler()),
])

# Categorical pipeline
cat_pipeline = Pipeline([
    # ("ordinal", OrdinalEncoder())  # Use this if you prefer ordinal encoding
    ("onehot", OneHotEncoder(handle_unknown="ignore"))
])

# Full pipeline
full_pipeline = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", cat_pipeline, cat_attribs),
])

# 6. Transform the data
housing_prepared = full_pipeline.fit_transform(housing)
```

```
# housing_prepared is now a NumPy array ready for training
print(housing_prepared.shape)
```

Training and Evaluating ML Models

Now that our data is preprocessed, let's move on to **training machine learning models** and evaluating their performance. We'll start with:

- **Linear Regression**
- **Decision Tree Regressor**
- **Random Forest Regressor**

We'll first test them on the training data and then use **cross-validation** to get a better estimate of their true performance.

---

1. Train and Test Models on the Training Set

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

# Linear Regression
lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)

# Decision Tree
tree_reg = DecisionTreeRegressor(random_state=42)
tree_reg.fit(housing_prepared, housing_labels)

# Random Forest
forest_reg = RandomForestRegressor(random_state=42)
forest_reg.fit(housing_prepared, housing_labels)

# Predict using training data
lin_preds = lin_reg.predict(housing_prepared)
```

```
tree_preds = tree_reg.predict(housing_prepared)
forest_preds = forest_reg.predict(housing_prepared)

# Calculate RMSE
lin_rmse = mean_squared_error(housing_labels, lin_preds, squared=False)
tree_rmse = mean_squared_error(housing_labels, tree_preds, squared=False)
forest_rmse = mean_squared_error(housing_labels, forest_preds, squared=False)

print("Linear Regression RMSE:", lin_rmse)
print("Decision Tree RMSE:", tree_rmse)
print("Random Forest RMSE:", forest_rmse)
```

A Warning About Training RMSE

Training RMSE **only shows how well the model fits the training data**. It does **not** tell us how well it will perform on unseen data. In fact, the **Decision Tree and Random Forest may overfit**, leading to very low training error but poor generalization.

---

2. Cross-Validation: A Better Evaluation Strategy

Cross-validation helps us evaluate how a model generalizes to new data without needing to touch the test set.

What is Cross-Validation?

Instead of training the model once and evaluating on a holdout set, **k-fold cross-validation** splits the training data into *k* folds (typically 10), trains the model on *k-1* folds, and validates it on the remaining fold. This process repeats *k* times.

We'll use `cross_val_score` from `sklearn.model_selection`.

Cross-Validation on Decision Tree

```
from sklearn.model_selection import cross_val_score
import pandas as pd

# Evaluate Decision Tree with cross-validation
tree_rmses = -cross_val_score(
    tree_reg,
    housing_prepared,
    housing_labels,
    scoring="neg_root_mean_squared_error",
    cv=10
)

# WARNING: Scikit-Learn’s scoring uses utility functions (higher is better), so
RMSE is returned as negative.
# We use minus (-) to convert it back to positive RMSE.
print("Decision Tree CV RMSEs:", tree_rmses)
print("\nCross-Validation Performance (Decision Tree):")
print(pd.Series(tree_rmses).describe())
```

Model Persistence and Inference with Joblib in a Random Forest Pipeline

Lets now summarize how to **train a Random Forest model on California housing data**, save the model and preprocessing pipeline using `joblib`, and reuse the model later for **inference on new data** (`input.csv`). This approach helps avoid retraining the model every time, improving performance and enabling reproducibility.

Why These Steps?

1. Why Train Once and Save?

- Training models repeatedly is **time-consuming** and **computationally expensive**.
- Saving the model (`model.pkl`) and preprocessing pipeline (`pipeline.pkl`) ensures you can **quickly load and run inference** anytime in the future.

2. Why Use a Preprocessing Pipeline?

- Raw data needs to be cleaned, scaled, and encoded before model training.
- A `Pipeline` automates this transformation and ensures **identical preprocessing** during inference.

3. Why Use Joblib?

- `joblib` efficiently serializes large NumPy arrays (like in sklearn models).
- Faster and more suitable than `pickle` for `scikit-learn` objects.

4. Why the If-Else Logic?

- The program checks if a saved model exists.
    - If **not**, it trains and saves the model.
    - If it **does**, it skips training and **only runs inference**, saving time.

Full Code

```
import os
import pandas as pd
import numpy as np
import joblib

from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.ensemble import RandomForestRegressor

MODEL_FILE = "model.pkl"
PIPELINE_FILE = "pipeline.pkl"

def build_pipeline(num_attribs, cat_attribs):
    num_pipeline = Pipeline([
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler())
    ])
    cat_pipeline = Pipeline([
        ("onehot", OneHotEncoder(handle_unknown="ignore"))
    ])
    full_pipeline = ColumnTransformer([
        ("num", num_pipeline, num_attribs),
        ("cat", cat_pipeline, cat_attribs)
    ])
    return full_pipeline
```

```
if not os.path.exists(MODEL_FILE):
    # TRAINING PHASE
    housing = pd.read_csv("housing.csv")
    housing['income_cat'] = pd.cut(housing["median_income"],
                                   bins=[0.0, 1.5, 3.0, 4.5, 6.0, np.inf],
                                   labels=[1, 2, 3, 4, 5])
    split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
    for train_index, _ in split.split(housing, housing['income_cat']):
        housing = housing.loc[train_index].drop("income_cat", axis=1)

    housing_labels = housing["median_house_value"].copy()
    housing_features = housing.drop("median_house_value", axis=1)

    num_attribs = housing_features.drop("ocean_proximity", axis=1).columns.tolist()
    cat_attribs = ["ocean_proximity"]

    pipeline = build_pipeline(num_attribs, cat_attribs)
    housing_prepared = pipeline.fit_transform(housing_features)

    model = RandomForestRegressor(random_state=42)
    model.fit(housing_prepared, housing_labels)

    # Save model and pipeline
    joblib.dump(model, MODEL_FILE)
    joblib.dump(pipeline, PIPELINE_FILE)

    print("Model trained and saved.")

else:
    # INFERENCE PHASE
    model = joblib.load(MODEL_FILE)
    pipeline = joblib.load(PIPELINE_FILE)

    input_data = pd.read_csv("input.csv")
    transformed_input = pipeline.transform(input_data)
    predictions = model.predict(transformed_input)
    input_data["median_house_value"] = predictions
```

```
    input_data.to_csv("output.csv", index=False)
    print("Inference complete. Results saved to output.csv")
```

Summary

With this setup, our ML pipeline is:

- **Efficient** – No retraining needed if the model exists.
- **Reproducible** – Same preprocessing logic every time.
- **Production-ready** – Can be deployed or reused across multiple systems.

Conclusion: California Housing Price Prediction Project

In this project, we built a complete machine learning pipeline to **predict California housing prices** using various regression algorithms. We started by:

- **Loading and preprocessing** the dataset (`housing.csv`) with careful treatment of missing values, scaling, and encoding using a custom pipeline.
- **Stratified splitting** was used to maintain income category distribution between train and test sets.
- We **trained and evaluated multiple algorithms** including:
    - Linear Regression
    - Decision Tree Regressor
    - Random Forest Regressor
- Through **cross-validation**, we found that **Random Forest performed the best**, offering the lowest RMSE and most stable results.

Finally, we built a script that:

- Trains the Random Forest model and saves it using `joblib` .
- Uses an **if-else logic** to skip retraining if the model exists.
- Applies the trained model to new data (`input.csv`) to predict `median_house_value` , storing results in `output.csv` .

This pipeline ensures that predictions are **accurate**, **efficient**, and **ready for production deployment**.