# Introduction to Scikit-Learn -by Mithun

---

## 1. Introduction

In Python, **scikit-learn (sklearn)** is the most popular library for implementing ML algorithms efficiently. It provides ready-to-use tools for:

- Loading datasets
- Splitting data
- Training ML models
- Evaluating performance
- Visualizing results

This report explains each step of a basic ML project using the **California Housing Dataset**.

---

## 2. Dataset and Objective

We use the **California Housing Dataset** from scikit-learn.
It contains information about California districts such as:

- Median income
- Average number of rooms
- Population
- Proximity to the ocean
  and the **target variable (y)** is the **median house price**.

**Objective:**
To build an ML model that predicts housing prices based on these features.

---

## 3. Step-by-Step ML Workflow

### Step 1: Import Required Libraries

We import essential libraries for dataset handling, model building, and visualization:

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt
```

- `datasets`: For loading sample data
- `model_selection`: For splitting and tuning models
- `preprocessing`: For scaling features
- `metrics`: For evaluating performance
- `matplotlib`: For plotting results

---

## Step 2: Load the Dataset

```
data = fetch_california_housing()
X, y = data.data, data.target
```

- `X` = Input features (independent variables)
- `y` = Output or target variable (dependent variable)

This gives us a **feature matrix (X)** and a **label vector (y)**.

---

## Step 3: Split the Dataset

We divide the dataset into **training** and **testing** parts.

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

- **Training set (80%)** → used to train the model.
- **Testing set (20%)** → used to evaluate how well the model generalizes.

***Why split?***
If we test on the same data we trained on, the model might “memorize” instead of learning — this is called **overfitting**.

---

## Step 4: Data Preprocessing (Scaling)

Different features may have different ranges (e.g., population vs. rooms).
To make sure no feature dominates others, we scale the data:

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

***Why scale?***
Algorithms like **KNN** and **SVM** are distance-based; scaling ensures fairness across all features.

---

## Step 5: Choose and Train a Model

We try two models:

**a) Linear Regression**

This assumes a straight-line relationship between input and output.

```
lr_model = LinearRegression()
lr_model.fit(X_train, y_train)
```

**b) K-Nearest Neighbors (KNN)**

This model predicts output based on the average of its nearest data points.

```
knn_model = KNeighborsRegressor(n_neighbors=5)
knn_model.fit(X_train, y_train)
```

***Why two models?***
Different algorithms behave differently on the same dataset.
Trying multiple models helps compare performance.

---

## Step 6: Make Predictions

Once trained, we test the models on unseen data:

```
y_pred_lr = lr_model.predict(X_test)
y_pred_knn = knn_model.predict(X_test)
```

- `y_pred_lr` → predictions from Linear Regression
- `y_pred_knn` → predictions from KNN

---

## Step 7: Evaluate Model Performance

We measure how close predictions are to real values.

```
print("Linear Regression:")
print("MSE:", mean_squared_error(y_test, y_pred_lr))
print("R² Score:", r2_score(y_test, y_pred_lr))

print("\nKNN Regressor:")
print("MSE:", mean_squared_error(y_test, y_pred_knn))
print("R² Score:", r2_score(y_test, y_pred_knn))
```

**Metrics Explained:**

- **MSE (Mean Squared Error):** Lower = better
- **R² Score:** Closer to 1 = better model fit

---

## Step 8: Visualize the Results

Plotting helps us visually check how accurate the predictions are.

```
plt.figure(figsize=(6,6))
plt.scatter(y_test, y_pred_lr, color="blue", alpha=0.5, label="Linear Regression")
plt.scatter(y_test, y_pred_knn, color="red", alpha=0.5, label="KNN")
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.legend()
plt.title("Actual vs Predicted Housing Prices")
plt.show()
```

***Interpretation:***
If the points are close to the diagonal line, predictions are accurate.

---

## Step 9: Hyperparameter Tuning using GridSearchCV

Instead of manually guessing the best `n_neighbors` for KNN,
we can use **GridSearchCV** to test multiple values automatically.

```
param_grid = {'n_neighbors': [3, 5, 7, 9]}
grid = GridSearchCV(KNeighborsRegressor(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best parameters:", grid.best_params_)
```

*How it works:*
GridSearchCV trains and tests models for each parameter combination using **cross-validation**, and picks the one with the best accuracy.

---

## 4. Summary of the Workflow

| Step | Task | Description |
|---|---|---|
| 1 | Import Libraries | Load sklearn, numpy, matplotlib, etc. |
| 2 | Load Dataset | Fetch California Housing data |
| 3 | Split Data | Divide into training/testing |
| 4 | Scale Data | Standardize feature values |
| 5 | Train Model | Fit Linear Regression or KNN |
| 6 | Predict | Generate predictions for test data |
| 7 | Evaluate | Check performance using metrics |
| 8 | Visualize | Plot actual vs predicted |
| 9 | Tune | Optimize hyperparameters with GridSearchCV |

---

## 5. Key Takeaways

- Scikit-learn offers a modular approach to machine learning.
- Always partition data to mitigate overfitting.
- Feature scaling is essential for distance-based algorithms.
- Employ multiple models to facilitate performance comparison.
- Hyperparameter tuning with GridSearchCV aids in identifying optimal configurations.
- Visualization is instrumental in interpreting model quality.

# PREPOCESSING

---

## 2. StandardScaler

### 2.1 Concept

The **StandardScaler** standardizes features by removing the mean and scaling to unit variance. It transforms each feature such that it has a mean of 0 and a standard deviation of 1.

Formula:
(X_{scaled} = \frac{X - \mu}{\sigma})

Where:

- (\mu) is the mean of the feature
- (\sigma) is the standard deviation

### 2.2 Limitation: Sensitivity to Outliers

StandardScaler is **not robust** to outliers. A single extreme value can distort the mean and standard deviation, leading to misleading scaling. For example, if one feature value is significantly higher than the others, the scaled data will be compressed around zero, and the model might underperform.

---

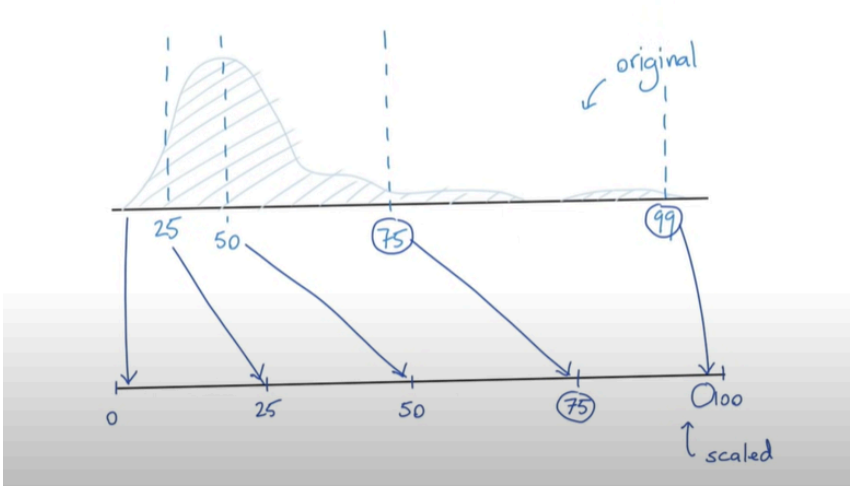
## 3. Quantiles and QuantileTransformer

### 3.1 Understanding Quantiles

Quantiles divide the data into equal-sized intervals. For instance:

- The 25th percentile (Q1) marks the point below which 25% of data lies.
- The 50th percentile (Q2 or median) marks the halfway point.

- The 75th percentile (Q3) indicates that 75% of data lies below it.




### 3.2 QuantileTransformer

The **QuantileTransformer** maps the original data distribution to a uniform or normal distribution based on quantiles. This technique helps to mitigate the influence of outliers and create a smoother, more balanced scale for features.

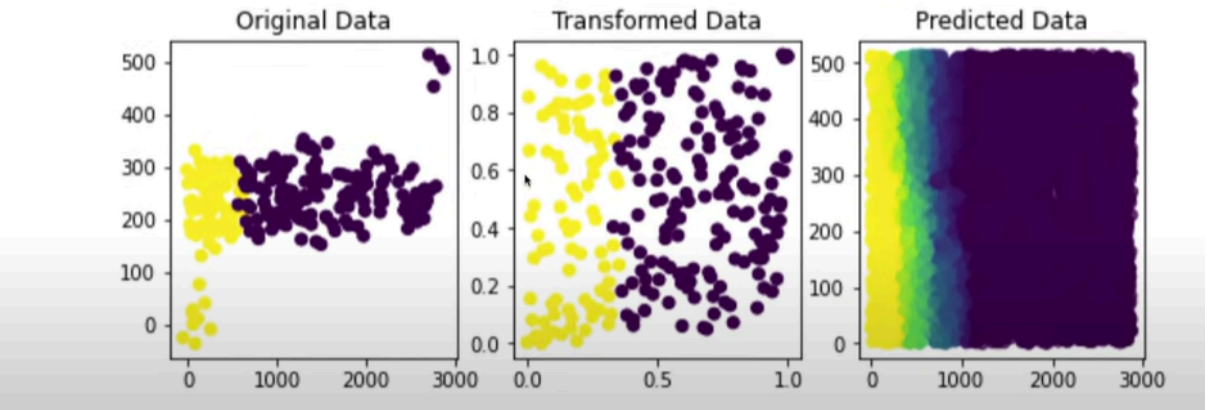
### 3.3 Visual Representation

The provided diagram demonstrates how quantile transformation works. Original data points (e.g., 25, 50, 75, 99) from a skewed distribution are mapped linearly to a new scale (0–100). This makes the transformed data uniformly spaced, removing skewness and reducing the impact of extreme values.

### 3.4 Use Cases

- When data contains **significant outliers**.
- When features follow a **non-Gaussian distribution**.
- For models sensitive to feature scaling, such as SVM or KNN.

```
[39]: plot_output(scaler=QuantileTransformer(n_quantiles=100))
```



---

# 4. Polynomial Features

## 4.1 Concept

Linear models can only capture straight-line relationships. However, many real-world problems involve nonlinear relationships between features and the target variable. **Polynomial features** extend linear models by adding nonlinear combinations of input features.

## 4.2 Example

If we have a single feature (x), the model can be expanded as:
(y = w_0 + w_1x + w_2x^2 + w_3x^3 + \ldots)

In scikit-learn, this is done using:

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X)
```

## 4.3 Benefit

Adding polynomial terms allows models such as **Linear Regression** or **Logistic Regression** to fit curved patterns, improving accuracy on datasets with nonlinear relationships.

---

# 5. OneHotEncoder

## 5.1 Purpose

Most machine learning models require numerical input. The **OneHotEncoder** converts categorical features into numerical form by creating binary columns (0 or 1) for each category.

## 5.2 Example

| Color | Red | Blue | Green |
|---|---|---|---|
| Red | 1 | 0 | 0 |
| Blue | 0 | 1 | 0 |
| Green | 0 | 0 | 1 |

In scikit-learn:

```
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder(sparse_output=False)
```

```
X_encoded = encoder.fit_transform(X)
```

### 5.3 Advantages

- Prevents numerical ordering in categorical data.
- Helps models understand distinct categories.
- Especially useful for tree-based and regression models.

### 5.4 Limitations

- May create many columns if a feature has numerous categories (known as the **curse of dimensionality**).
- Does not handle unknown categories unless explicitly configured.

---

## 6. Summary

| Technique | Purpose | Best Used When | Limitation |
|---|---|---|---|
| StandardScaler | Standardize features to mean=0, std=1 | Data is approximately normal and outliers are minimal | Sensitive to outliers |
| QuantileTransformer | Map data to uniform/normal distribution | Data is skewed or contains outliers | May distort small datasets |
| PolynomialFeatures | Add nonlinear feature combinations | When data relationships are nonlinear | May cause overfitting |
| OneHotEncoder | Encode categorical variables | When data has text categories | Increases feature space size |

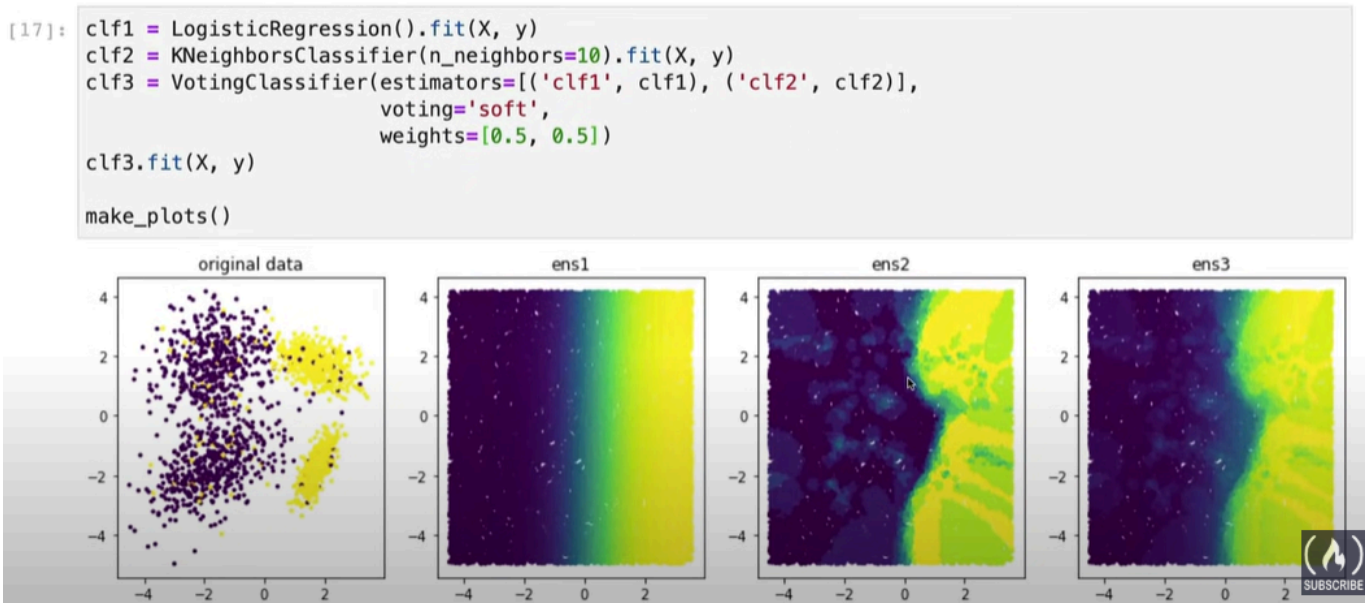
---

## 7. Conclusion

Data preprocessing is the foundation of effective machine learning. Choosing the right preprocessing method depends on the data distribution, feature type, and model sensitivity. Techniques like **QuantileTransformer** make data robust to outliers, while **PolynomialFeatures** expand model capability to capture complex patterns. **OneHotEncoder** ensures categorical variables are properly represented, improving model interpretability and accuracy.

# METRICS


# META ESTIMATORS




# Meta-Estimators in Scikit-learn

# 1. Introduction

Meta-estimators are models that build upon other models. Instead of relying on a single algorithm, they combine the predictions of multiple estimators to produce more robust and accurate results. Scikit-learn provides several meta-estimators, such as **VotingClassifier**, **BaggingClassifier**, and **StackingClassifier**. These methods enhance model stability, reduce overfitting, and balance performance across datasets.

---

# 2. Voting Classifier

## 2.1 Concept

The **VotingClassifier** is a simple ensemble technique that aggregates predictions from multiple models (also called base estimators). The final prediction is determined either by majority voting or by averaging probabilities.

- **Hard Voting:** Each model votes for a class label, and the class with the most votes becomes the final prediction.
- **Soft Voting:** Each model predicts class probabilities, and the final prediction is based on the average of these probabilities.

This method assumes that combining several weak or moderately strong models can lead to a more reliable and stable prediction overall.

## 2.2 Example

```
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC

log_clf = LogisticRegression()
knn_clf = KNeighborsClassifier()
svc_clf = SVC(probability=True)

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('knn', knn_clf), ('svc', svc_clf)],
    voting='soft'
)
```

The VotingClassifier combines the strengths of Logistic Regression, KNN, and SVM to make a collective decision.

## 2.3 Difference Between Logistic Regression and KNN

| Feature | Logistic Regression | K-Nearest Neighbors |
|---|---|---|
| **Type** | Parametric (learns weights) | Non-parametric (memory-based) |
| **Learning Approach** | Learns a decision boundary during training | Makes predictions based on nearest data points |
| **Complexity** | Fast to train, slower to converge for nonlinear data | Slow at prediction time due to distance calculations |
| **Interpretability** | Provides coefficients and probabilities | Difficult to interpret, depends on data distribution |
| **Usage in VotingClassifier** | Learns a global relationship | Adds local neighborhood-based decisions |

The difference in nature between these models helps improve ensemble diversity, which is key to a strong voting classifier.

---

# 3. Scikit-Lego

## 3.1 Overview

**Scikit-lego** is an extension library for scikit-learn that introduces additional transformers, estimators, and tools designed for more flexible workflows. It complements scikit-learn by providing practical utilities that are often needed in real-world machine learning pipelines.

## 3.2 Features

- Extra meta-estimators and pipeline utilities
- Transformers for data cleaning and feature engineering
- Model inspection tools and wrappers

Example usage:

```
from sklego.meta import EstimatorTransformer
from sklearn.linear_model import LogisticRegression

model = EstimatorTransformer(LogisticRegression())
```

This library enables scikit-learn users to extend standard functionality without rewriting core components.

---

# 4. HumanLearn

### 4.1 Concept

**HumanLearn** is a library built to bridge the gap between human reasoning and machine learning models. It allows the creation of interpretable rules based on human-understandable logic rather than complex algorithms.

### 4.2 Purpose

The main objective of HumanLearn is to make models explainable, auditable, and easier to understand. It focuses on rule-based decision making rather than opaque statistical fitting.

Example:

```
from humanlearn import RuleLearner
rules = [
    (lambda x: x['age'] > 50, 1),
    (lambda x: x['income'] < 20000, 0)
]
model = RuleLearner(rules=rules)
```

This provides a structured way to express model logic similar to human decision-making.

---

# 5. Conclusion

Meta-estimators in scikit-learn represent a shift from relying on single models to combining multiple approaches for greater reliability. The **VotingClassifier** exemplifies this by merging diverse algorithms like Logistic Regression and KNN, improving performance and generalization. Tools such as **scikit-lego** and **HumanLearn** extend scikit-learn's capabilities, promoting more expressive, interpretable, and adaptable workflows for modern machine learning tasks.