Scikit-Learn: The Ultimate Syntax Guide

Scikit-Learn (`sklearn`) is the industry standard for machine learning in Python. Its power lies in its **consistency**: almost every algorithm follows the exact same "Estimator API".

1. The Mental Model

The Estimator API

Memorize this workflow, and you know how to use 90% of the library.

```
graph LR
    A["Instantiate Model"] --> B["Fit (Learn)"]
    B --> C{"Predict or Transform?"}
    C -->|Supervised| D["Predict (Apply)"]
    C -->|Preprocessing| E["Transform (Modify)"]
```

1. **Instantiate**: Create the model object (e.g., `model = LinearRegression()`).
2. **Fit**: The model **learns** patterns from the training data.
3. **Predict**: The model **applies** what it learned to new data (Supervised Learning).
4. **Transform**: The model **modifies** the data based on what it learned (Preprocessing).

The Data Shape

- **X (Features)**: A 2D Matrix (Rows = Samples, Cols = Features).
- **y (Target)**: A 1D Vector (Labels).

---

2. Universal Syntax Patterns

Pattern 1: Supervised Learning (Predict)

Goal: Predict a target `y` from features `X`.

```python
# 1. Import & Instantiate
from sklearn.module import ModelName
model = ModelName(hyperparameter=value)

# 2. Fit (Train on labeled data)
model.fit(X_train, y_train)

# 3. Predict (Test on new data)
predictions = model.predict(X_test)
```

Pattern 2: Preprocessing (Transform)

Goal: Modify `X` to make it suitable for ML.

```
# 1. Import & Instantiate
from sklearn.module import TransformerName
scaler = TransformerName()

# 2. Fit (Learn stats like mean/std from Training Data)
scaler.fit(X_train)

# 3. Transform (Apply to both Train and Test)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

> [!IMPORTANT] **Golden Rule**: NEVER `fit` on your test data. Only `transform` it. This prevents "Data Leakage" (cheating).

---

3. Preprocessing (The Foundation)

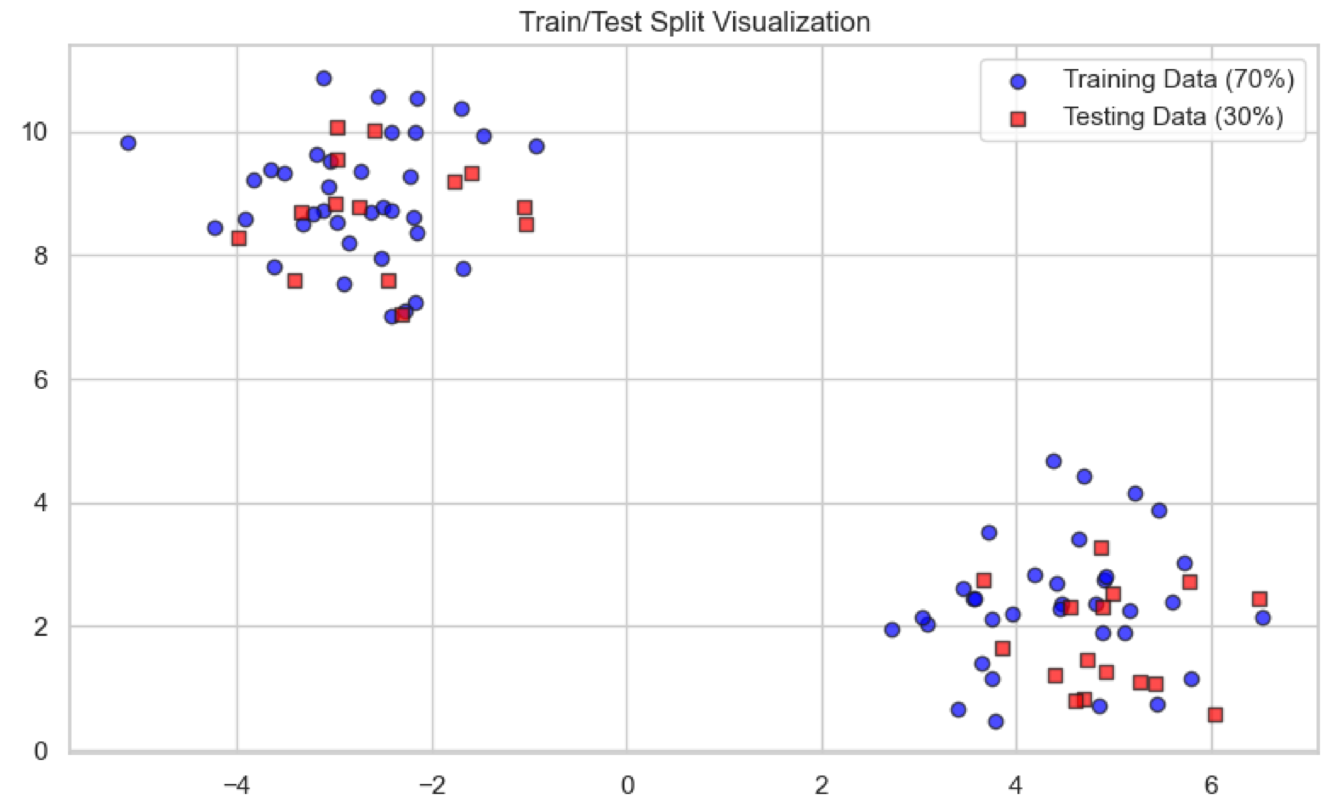
Data must be numerical and clean before ML.

Splitting Data

Why? To simulate how the model performs on unseen data.

```
from sklearn.model_selection import train_test_split

# 80% Train, 20% Test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

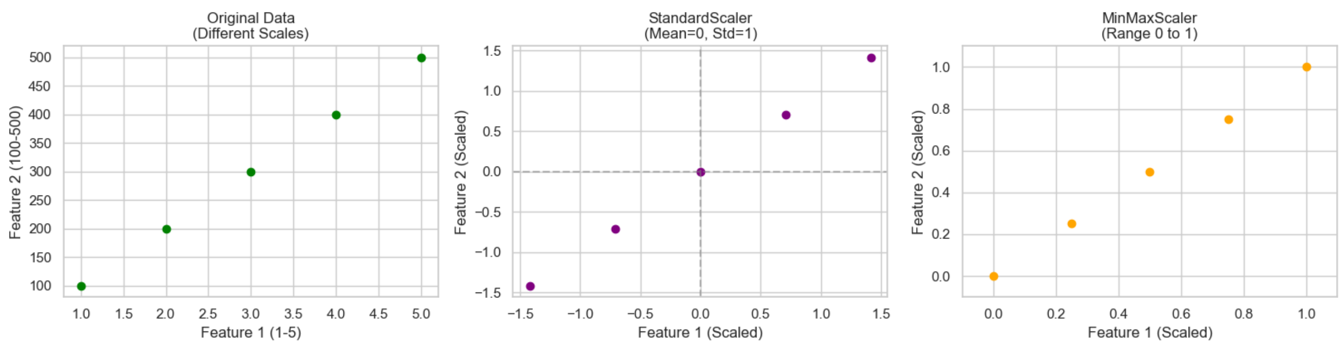


Scaling (Normalization)

Why? Models like KNN and SVM calculate distances. If one feature is in millions and another in decimals, the big one dominates.

Scaler	Intuition	Formula
StandardScaler	"Bell Curve". Centers data around 0.	$z = \frac{x - \mu}{\sigma}$
MinMaxScaler	"Squeeze". Compresses data to [0, 1].	$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$

```python
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X) # Fit & Transform in one step
```

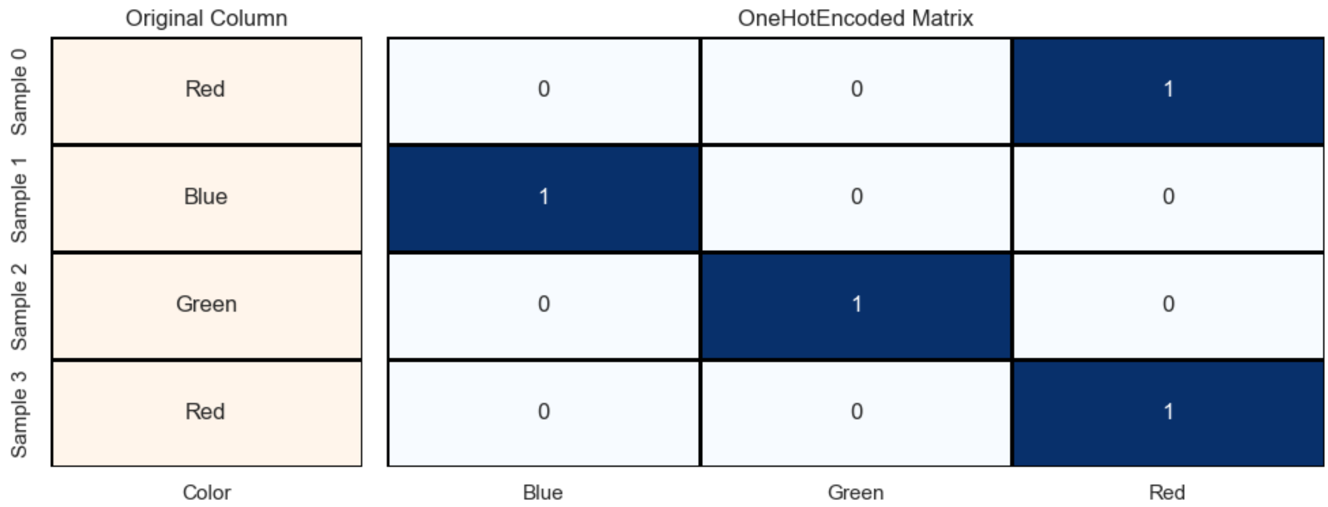


Encoding (Categorical to Numerical)

Why? Models only understand numbers, not strings like "Red" or "Blue".

Encoder	Use Case
OneHotEncoder	Nominal Features (Red, Blue). Creates new binary columns.
LabelEncoder	Target Labels (Cat, Dog). Converts to 0, 1.

```
# OneHotEncoder for Features (X)
from sklearn.preprocessing import OneHotEncoder
enc = OneHotEncoder(sparse_output=False)
X_encoded = enc.fit_transform(X_categorical)
```



---

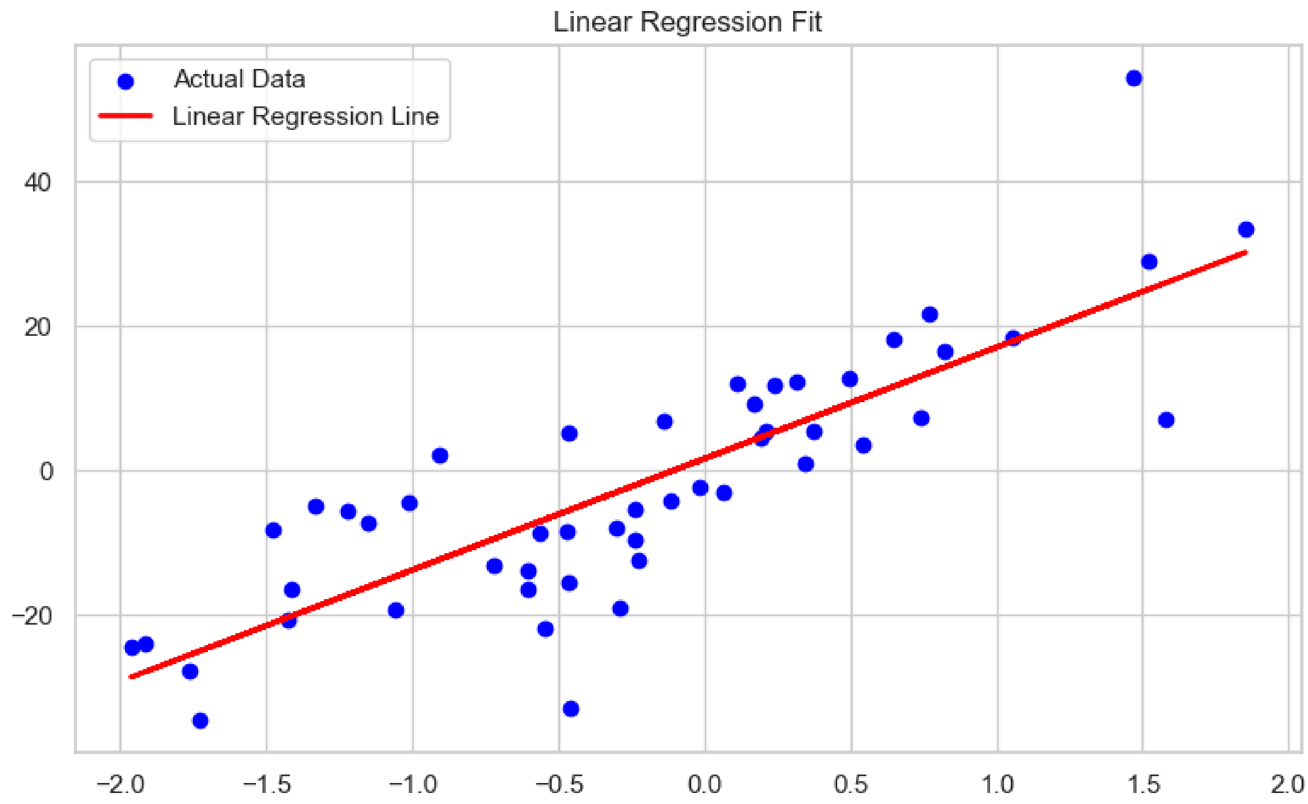
4. Supervised Learning Reference

Regression (Predicting Numbers)

Target: Continuous value (Price, Temperature).

Algorithm	Import Path	Key Hyperparameters
Linear Regression	`sklearn.linear_model`	None usually
Random Forest Regressor	`sklearn.ensemble`	`n_estimators` (trees)

```
from sklearn.linear_model import LinearRegression
reg = LinearRegression()
reg.fit(X_train, y_train)
```

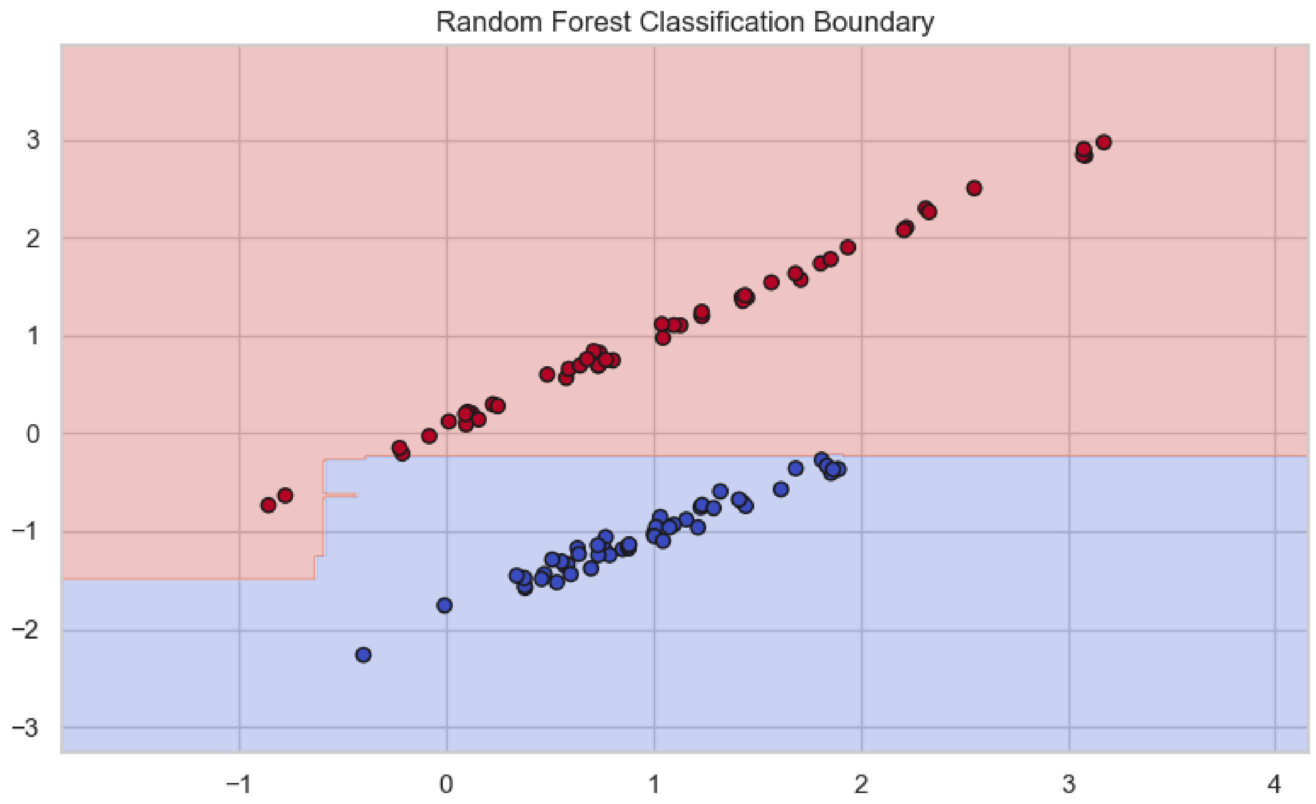


Classification (Predicting Categories)

Target: Discrete Class (Spam/Not Spam, Cat/Dog).

Algorithm	Import Path	Key Hyperparameters
Logistic Regression	`sklearn.linear_model`	`C` (Regularization)
Random Forest Classifier	`sklearn.ensemble`	`n_estimators`

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier()
clf.fit(X_train, y_train)
```



---

5. Model Evaluation

Metrics

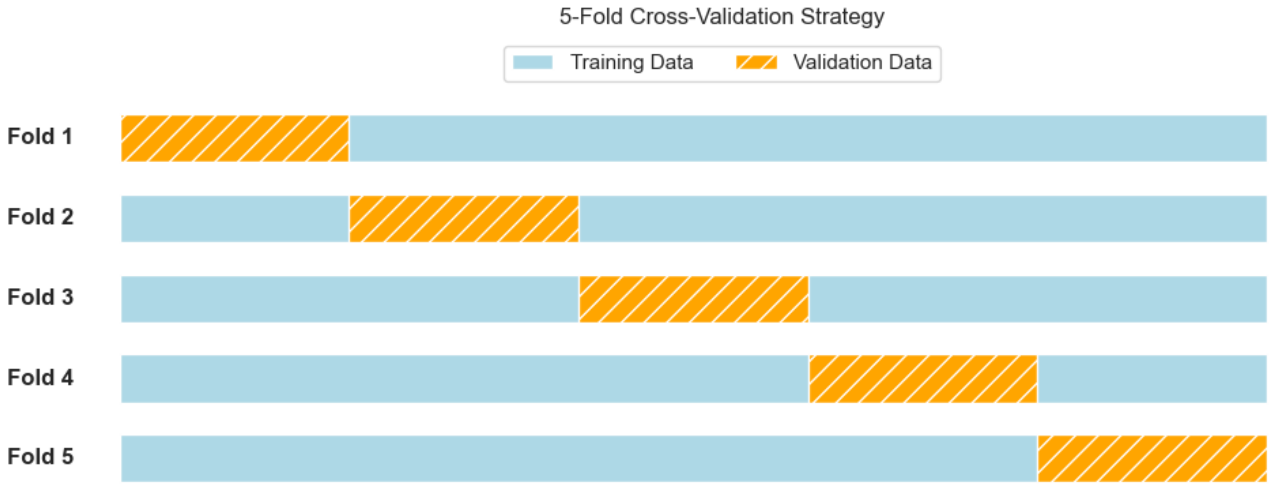
How good is the model?

Task	Metric	Function
Classification	Accuracy	`accuracy_score(y_true, y_pred)`
Regression	Error (MSE)	`mean_squared_error(y_true, y_pred)`

Cross-Validation

Intuition: "Test on multiple folds to be sure."

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(model, X, y, cv=5)
print(f"Average Accuracy: {scores.mean()}")
```



---

6. Hyperparameter Tuning with GridSearchCV

The Problem: Manual Tuning is Inefficient

Every model has **hyperparameters** (settings you choose) that affect performance.

```
# Manual tuning = exhausting!
model1 = LogisticRegression(C=0.1)
model2 = LogisticRegression(C=1.0)
model3 = LogisticRegression(C=10)
# ... compare all 3? What if you have 5 hyperparameters?
```

The Solution: GridSearchCV

Intuition: "Automatically test all combinations of hyperparameters and pick the best."

Syntax: `GridSearchCV(estimator, param_grid, cv=5)`

- **estimator**: The model to tune (e.g., `LogisticRegression()`)
- **param_grid**: Dictionary of hyperparameters to test
- **cv**: Number of cross-validation folds

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# Define hyperparameter grid
param_grid = {
    'C': [0.1, 1, 10, 100],
    'penalty': ['l2', 'l1'],
    'max_iter': [100, 500]
}

# Create GridSearchCV
grid_search = GridSearchCV(
```

```python
    estimator=LogisticRegression(solver='liblinear'),
    param_grid=param_grid,
    cv=5,  # 5-fold cross-validation
    scoring='accuracy'
)

# Fit (tests all combinations)
grid_search.fit(X_train, y_train)

# Get results
print(f"Best Parameters: {grid_search.best_params_}")
print(f"Best Score: {grid_search.best_score_}")

# Use best model
best_model = grid_search.best_estimator_
predictions = best_model.predict(X_test)
```

Key Attributes

- `best_params_`: The hyperparameter combination that scored highest
- `best_score_`: The cross-validation score of the best combination
- `best_estimator_`: The fitted model using best parameters (ready to predict!)
- `cv_results_`: DataFrame with all results for detailed analysis

Example: Access All Results

```python
import pandas as pd

# Get all results as DataFrame
results_df = pd.DataFrame(grid_search.cv_results_)
results_df[['param_C', 'param_penalty',
'mean_test_score']].sort_values('mean_test_score', ascending=False)
```