

WORKFLOW

```
raw data
-> train_test_split
-> Pipeline[ preprocess -> model ]
-> fit(X_train, y_train)
-> predict(X_test)
-> metrics / cross-validation
-> tune (GridSearchCV)
-> final check on test set
-> joblib.dump
```

Naming: **X** = features (2D), **y** = target (1D).

ESTIMATOR API

<code>model.fit(X, y)</code>	learn from training data
<code>model.predict(X)</code>	predicted labels / values
<code>model.predict_proba(X)</code>	class probabilities (n, classes)
<code>t.fit(X)</code> <code>t.transform(X)</code>	transformers: learn, then apply
<code>t.fit_transform(X_train)</code>	both steps (training data only)
<code>model.score(X, y)</code>	R2 (regression) / accuracy (classif.)
<code>model.get_params()</code>	list hyperparameters

Learned attributes end in `_`, e.g. `coef_`.

TRAIN / TEST SPLIT

```
from sklearn.model_selection import (
    train_test_split)
X_tr, X_te, y_tr, y_te = train_test_split(
    X, y, test_size=0.2,
    random_state=42,
    stratify=y) # keep class ratios
```

Use `stratify=y` for classification only.

PREPROCESSING

<code>StandardScaler()</code>	mean 0, std 1 per feature
<code>MinMaxScaler()</code>	scale to [0, 1]
<code>OneHotEncoder(handle_unknown="ignore")</code>	categories to 0/1 columns; unseen = all zeros
<code>SimpleImputer(strategy="median")</code>	fill NaN: mean, median, most_frequent, constant
<code>OrdinalEncoder()</code>	categories to integers

COLUMNTRANSFORMER + PIPELINE

```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import (
    Pipeline, make_pipeline)
num = make_pipeline(
    SimpleImputer(strategy="median"),
    StandardScaler())
cat = OneHotEncoder(handle_unknown="ignore")
pre = ColumnTransformer([
    ("num", num, ["age", "income"]),
    ("cat", cat, ["city"])]])
pipe = Pipeline([
    ("pre", pre),
    ("model", LogisticRegression())])
pipe.fit(X_tr, y_tr)
pipe.predict(X_te)
```

`make_pipeline` names steps automatically (lowercase class name).

MODELS: REGRESSION

<code>LinearRegression()</code>	ordinary least squares
<code>Ridge(alpha=1.0)</code>	L2 penalty; shrinks coefficients
<code>Lasso(alpha=0.1)</code>	L1 penalty; zeroes some coefficients
<code>RandomForestRegressor()</code>	averaged trees; robust
<code>KNeighborsRegressor()</code>	average of k nearest; scale first
<code>HistGradientBoostingRegressor()</code>	fast boosting; handles NaN (also GradientBoostingRegressor)

MODELS: CLASSIFICATION

<code>LogisticRegression()</code>	linear; probabilities; <code>max_iter=1000</code> if no converge
<code>KNeighborsClassifier()</code>	vote of k nearest (<code>n_neighbors=5</code>); scale first
<code>DecisionTreeClassifier()</code>	if/else rules; limit <code>max_depth</code>
<code>RandomForestClassifier()</code>	many trees (<code>n_estimators=200</code>); strong default
<code>HistGradientBoostingClassifier()</code>	fast boosting; handles NaN (also GradientBoostingClassifier)
<code>SVC()</code>	margin-based; scale first; <code>probability=True</code> for proba

UNSUPERVISED

<code>KMeans(n_clusters=3)</code>	clusters; <code>labels_</code> , <code>inertia_</code>
<code>PCA(n_components=2)</code>	reduce dimensions (scale first)
<code>pca.explained_variance_ratio_</code>	variance kept per component
<code>silhouette_score(X, labels)</code>	cluster quality, -1 to 1

REGRESSION METRICS

```
from sklearn.metrics import (
    mean_absolute_error, r2_score,
    root_mean_squared_error,
    mean_squared_error)
mean_absolute_error(y_te, pred) # MAE
root_mean_squared_error(y_te, pred) # RMSE
mean_squared_error(y_te, pred) # MSE
r2_score(y_te, pred) # R2 (1 = perfect)
```

`mean_squared_error(..., squared=False)` was removed; use `root_mean_squared_error` (1.4+).

CLASSIFICATION METRICS

```
from sklearn.metrics import *
accuracy_score(y_te, pred)
precision_score(y_te, pred) # TP / (TP+FP)
recall_score(y_te, pred) # TP / (TP+FN)
f1_score(y_te, pred)
confusion_matrix(y_te, pred) # rows=true
classification_report(y_te, pred)
proba = model.predict_proba(X_te)[: , 1]
roc_auc_score(y_te, proba) # needs scores
```

Multiclass: add `average="macro"` or `"weighted"`. With imbalance, rely on F1/recall, not accuracy.

CROSS-VALIDATION

```
from sklearn.model_selection import (
    cross_val_score, KFold, StratifiedKFold)
cv = StratifiedKFold(5, shuffle=True,
    random_state=42)
scores = cross_val_score(pipe, X, y, cv=cv,
    scoring="f1")
print(scores.mean(), scores.std())
```

`KFold` for regression, `StratifiedKFold` for classification.
Scoring: `"accuracy"`, `"roc_auc"`, `"r2"`, `"neg_root_mean_squared_error"` (higher = better, so negated).

HYPERPARAMETER TUNING

```
from sklearn.model_selection import (
    GridSearchCV)
grid = {"model__C": [0.1, 1, 10],
    "pre__num__simpleimputer__strategy":
        ["mean", "median"]}
gs = GridSearchCV(pipe, grid, cv=5,
    scoring="f1", n_jobs=-1)

gs.fit(X_tr, y_tr)
gs.best_params_   gs.best_score_
gs.best_estimator_.predict(X_te)
```

Param name = `stepname__param`.
`RandomizedSearchCV`(pipe, dists, `n_iter=20`) samples instead of trying all combinations.

INSPECT & SAVE

<code>rf.feature_importances_</code>	tree-model importances (sum to 1)
<code>lin.coef_</code> , <code>lin.intercept_</code>	linear model weights
<code>pipe[-1].get_feature_names_out()</code>	column names after preprocessing
<code>joblib.dump(pipe, "model.joblib")</code>	save whole pipeline
<code>joblib.load("model.joblib")</code>	load it back

Load models only from trusted files; keep the same scikit-learn version.

AVOID DATA LEAKAGE

Split **first**, then fit scalers/imputers/encoders on the training set only.
Put preprocessing **inside** the Pipeline so cross-validation refits it per fold.
Never tune on the test set; use CV, then test once.
Drop features that encode the target or the future.
Set `random_state` for reproducible results.