Lesson 32: Complete ML Pipeline

Data Science with Python – BSc Course

Data Science Program

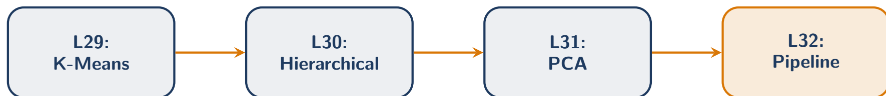
BSc Course

45 Minutes

Previously: Unsupervised Toolkit

Module 6 so far: three ways to find structure without labels.

- L29 K-Means – partition data into K flat clusters
- L30 Hierarchical – build a tree of nested clusters
- L31 PCA – compress dimensions while keeping variance



Now: assemble all the pieces into a single production machine.

Module 6 capstone – from individual tools to a complete assembly line

Danger of Doing It Wrong

Situation: You build a credit-risk model. Test accuracy: 99%.
You deploy it. Within a week, predictions are random.

Complication: What went wrong?

- You scaled ALL data (train + test) together before splitting
- The scaler's mean and std "knew" test-set statistics
- Model performance was inflated by **data leakage**

Question: How do we build ML workflows that *prevent* leakage by construction – not just by discipline?

Answer: Pipelines – an assembly line where each step only sees training data, automatically.

99% accuracy that collapses in production is worse than 80% accuracy that holds

What is Data Leakage?

Information from the future sneaking into training

- Scale all data first, then split $\rightarrow$ test statistics leak into scaler
- Shuffle time series before splitting $\rightarrow$ future leaks into past
- Pipeline solution: fit transformers *only* on training fold

Data leakage = seeing the exam answers before the test. Pipelines prevent it automatically.

Learning Objectives

After this lesson, you will be able to:

1. Build sklearn **Pipelines** that chain preprocessing and modeling into a single, leak-free object
2. Apply **cross-validation** correctly using pipelines (no data leakage)
3. Tune hyperparameters with **GridSearchCV** and **RandomizedSearchCV**
4. Handle financial time series with **TimeSeriesSplit** and the gap parameter

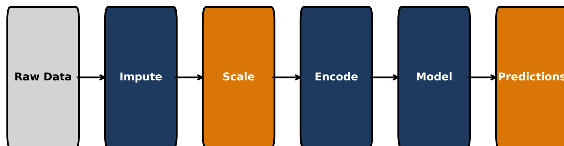
Mental model: Think of a pipeline as a factory conveyor belt – raw data enters one end, predictions exit the other. Every transformation is a station on the belt; nothing touches the test data.

Pipeline = the assembly line that turns raw data into predictions, safely

The Assembly Line Idea

Pipeline = sequence of transformers + final estimator

- Each step's output becomes the next step's input
- `pipe.fit(X_train, y_train)` fits all steps sequentially
- `pipe.predict(X_test)` transforms then predicts



One object, one `.fit()`, one `.predict()` – no manual step-chaining

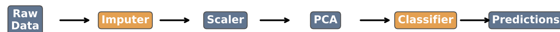
Pipeline Flow

Data flows through each station on the conveyor belt

- All steps except last must implement `transform()`
- Last step can be any estimator (`predict()`)

Pipeline Flow

fit_transform() / transform()



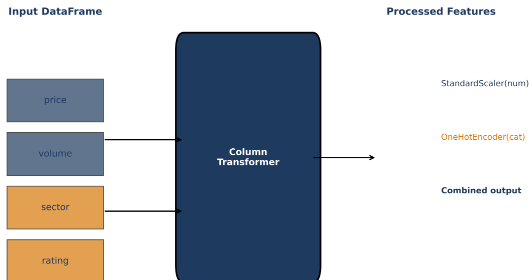
Pipeline chains all steps into ONE estimator

Raw $\longrightarrow$ impute $\longrightarrow$ scale $\longrightarrow$ reduce $\longrightarrow$ predict

ColumnTransformer

Different columns need different preprocessing

- Numeric (price, volume) $\rightarrow$ impute + scale
- Categorical (sector, exchange) $\rightarrow$ impute + encode

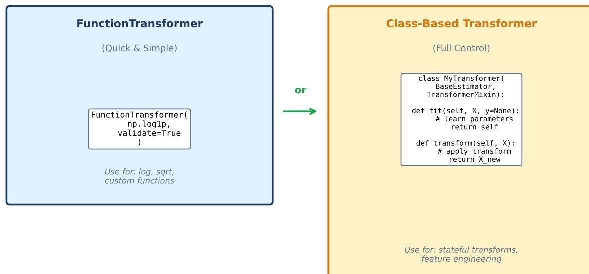


ColumnTransformer = "apply this to these columns, that to those"

Custom Transformers

Writing Your Own Transformer

- Inherit from `BaseEstimator, TransformerMixin`
- Implement `fit()` and `transform()` methods
- Enables domain-specific feature engineering inside a pipeline

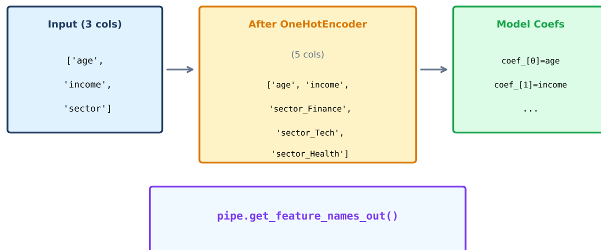


Custom transformers keep domain logic inside the pipeline – no separate scripts

Feature Names Tracking

Knowing what comes out of the pipeline

- `pipe[:-1].get_feature_names_out()` returns column names
- Essential for interpreting model coefficients after encoding/PCA

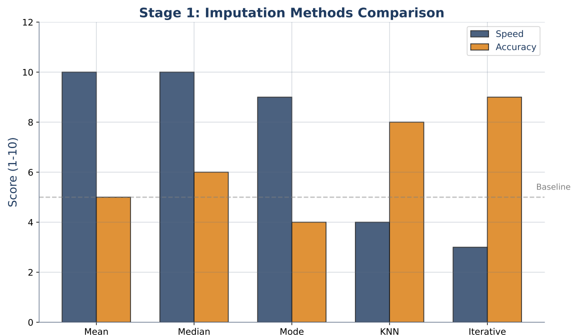


Feature name tracking prevents the "which column is which?" mystery

Preprocessing Steps

Step 1 – Imputation: fill missing values first

- `SimpleImputer`: fill with mean, median, or constant
- `KNNImputer`: fill based on nearest neighbors

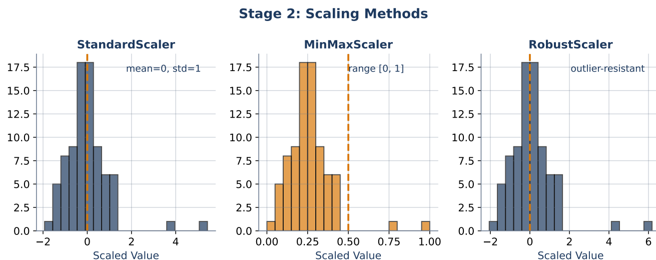


Pipeline order matters: impute ⟶ scale ⟶ encode ⟶ select

Step 2: Scaling

Normalize feature ranges

- `StandardScaler`: zero mean, unit variance
- `MinMaxScaler`: map to [0, 1] range
- Critical for distance-based methods (K-Means, PCA, SVM)

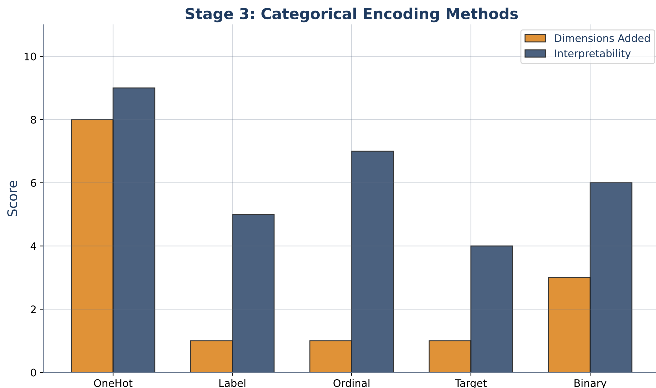


StandardScaler (mean=0, std=1) or MinMaxScaler (0–1 range)

Step 3: Encoding

Convert categories to numbers

- `OneHotEncoder`: nominal categories (sector, exchange)
- `OrdinalEncoder`: ordered categories (credit rating)

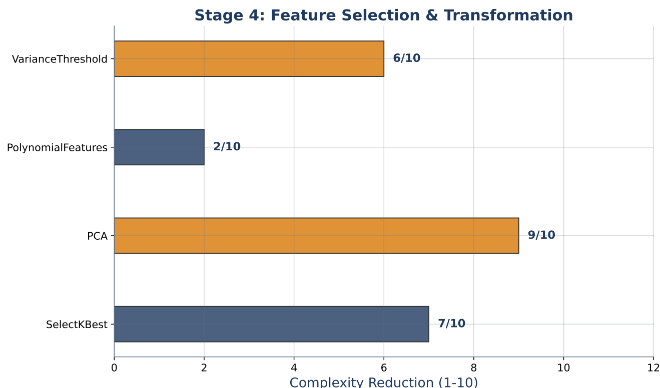


OneHotEncoder for nominal, OrdinalEncoder for ordinal categories

Step 4: Feature Selection / Transformation

Reduce or reshape the feature space

- `SelectKBest`: keep top K features by statistical test
- `PCA`: compress into principal components (see L31)



SelectKBest, PCA, or custom transformers for feature engineering

Why Cross-Validation?

One train/test split = a coin flip

The problem:

- Your score depends on *which* data landed in the test set
- Lucky split $\rightarrow$ overoptimistic; unlucky $\rightarrow$ pessimistic
- Result: unreliable performance estimate

The cross-validation fix:

- Split data K ways; rotate which fold is the test set
- Every data point is tested exactly once
- Average K scores $\rightarrow$ robust estimate; std reveals stability

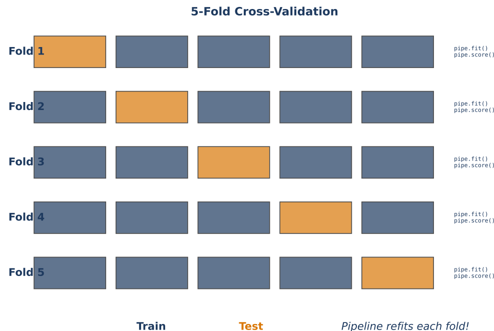
Key insight: CV estimates how well the model *generalizes* – it does **not** train the final model.

CV = multiple train/test splits to get a stable performance estimate

K-Fold Cross-Validation

Each fold serves as test set exactly once

- 5-fold: 80% train / 20% test per iteration
- `cross_val_score(pipe, X, y, cv=5)` returns 5 scores

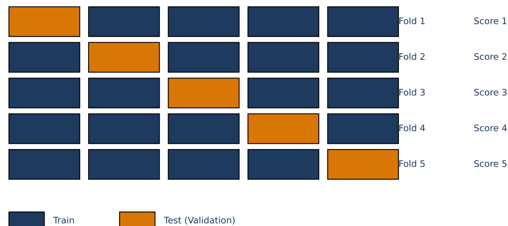


All data used for both training and testing – no wasted samples

CV with Pipeline = No Leakage

Why the pipeline matters inside CV

- Without pipeline: scale all data, *then* split → leakage
- With pipeline: scaler fits only on training fold each time
- `cross_val_score(pipe, X, y, cv=5)` handles this automatically

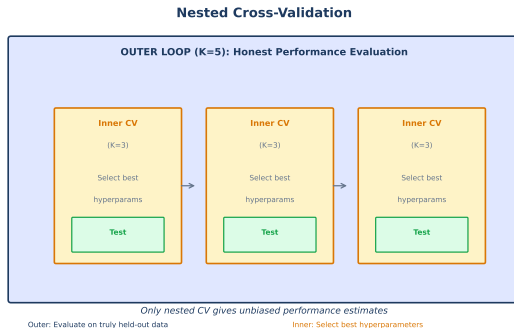


Pipeline inside CV = transformers refit on every training fold (no leakage)

Nested Cross-Validation

Tune and evaluate without bias

- Outer loop: evaluate generalization (5-fold)
- Inner loop: tune hyperparameters (3-fold per outer fold)

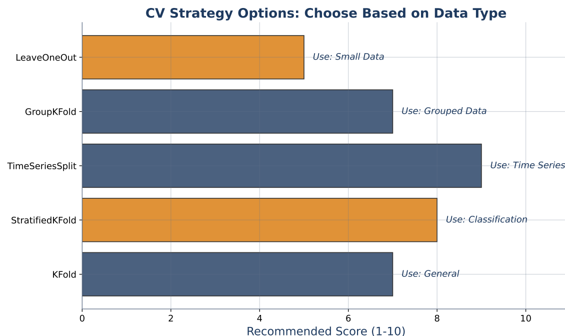


Nested CV: inner loop tunes, outer loop evaluates – no selection bias

CV Strategy Options

Choose the right splitter for your data

- `KFold` / `StratifiedKFold` / `GroupKFold`

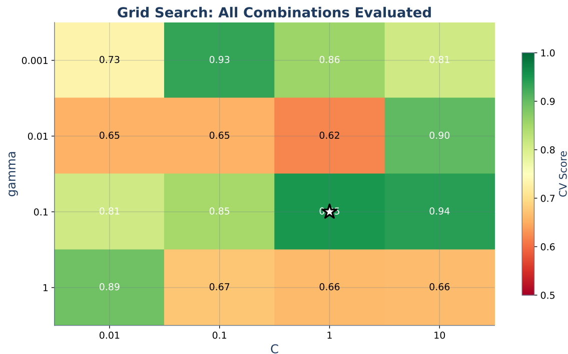


KFold, StratifiedKFold (classification), GroupKFold (grouped data)

Grid Search

Try every combination, pick the best

- Define grid: {`'model__alpha': [0.1, 1, 10]`}
- GridSearchCV scores all combinations via CV

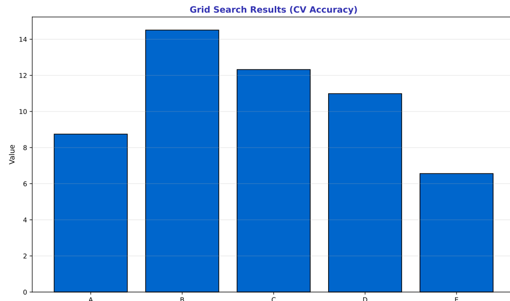


Grid = Cartesian product of all parameter values, scored by CV

Grid Search Results

Heatmap of scores across parameter combinations

- Each cell = mean CV score for that (alpha, n_components) pair
- Bright = high score; dark = low score

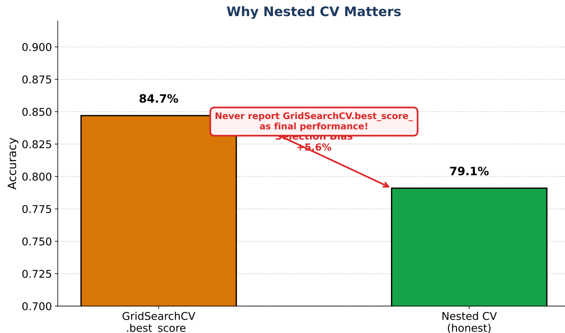


Heatmap reveals interaction effects between hyperparameters

Bias Without Nested CV

Tuning and evaluating on the same CV overestimates

- "Best" score is biased upward (selection effect)
- Nested CV gives an unbiased estimate of tuned models

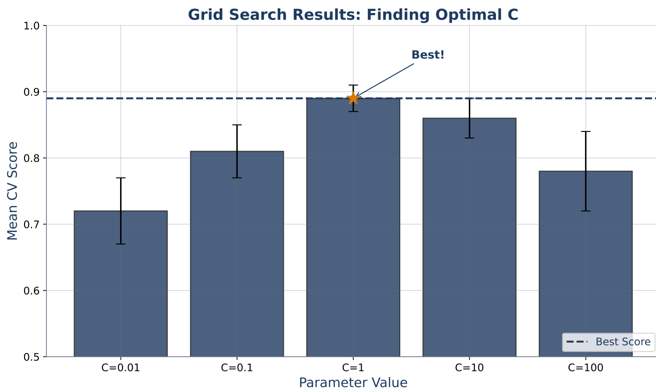


Without nested CV, reported accuracy is typically 2–5% too optimistic

Results Analysis

Mining `cv_results_` for insights

- `grid.cv_results_` is a dict with scores, times, and parameters
- Convert to DataFrame for analysis: rank, mean/std of test scores

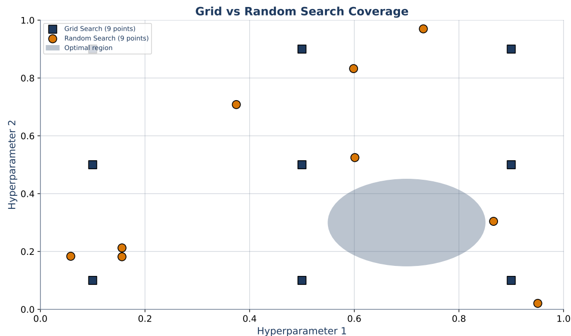


`cv_results_` DataFrame contains all scores and parameters for every combination

Random Search

Sample randomly instead of exhaustive grid

- Grid grows exponentially; random: set budget (`n_iter`)
- Often finds equally good parameters with far less compute

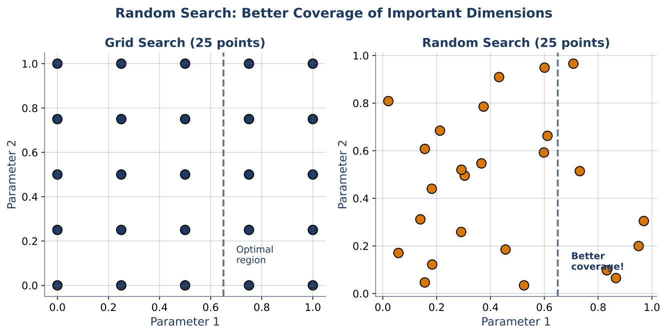


Random search covers more of each dimension – better for large spaces

Why Random Search Works

Not all parameters matter equally

- Grid wastes trials on unimportant parameter values
- Random explores more unique values per dimension
- With 60 random trials, 95% chance of hitting top-5% region

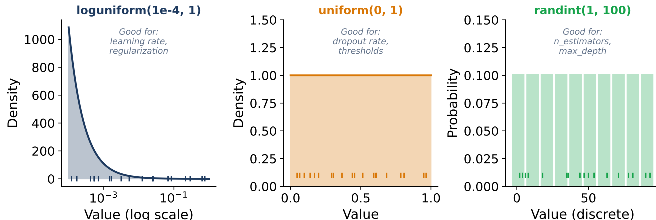


Random covers more of each dimension – better for unimportant params

Distribution Sampling

Choose the right distribution for each hyperparameter

- `loguniform(1e-4, 1)`: regularization, learning rate, SVM C
- `uniform(0, 1)`: dropout rate, split threshold
- `randint(5, 50)`: number of components, neighbors, depth

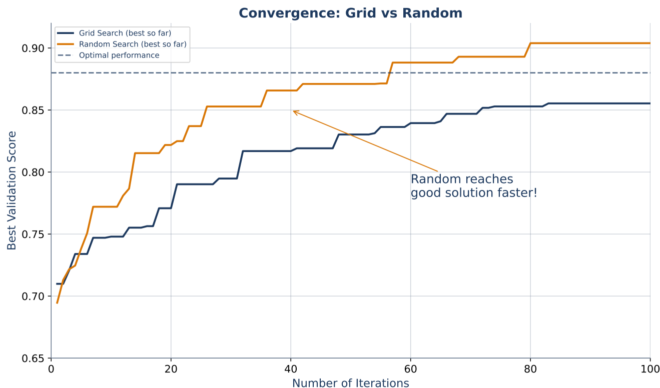


loguniform = fair sampling across orders of magnitude (0.001, 0.01, 0.1, 1)

Convergence: Grid vs Random

Random often finds good parameters faster

- With the same compute budget, random typically wins
- Advantage grows with number of hyperparameters

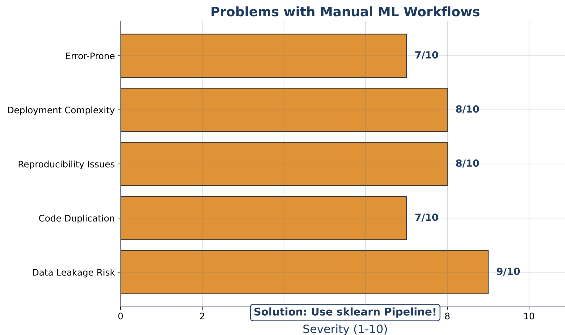


Random often finds good params faster than grid with same budget

Finance: The Time Problem

Standard CV shuffles data – disaster for time series

- Shuffle lets future returns leak into training
- In finance, temporal order is everything

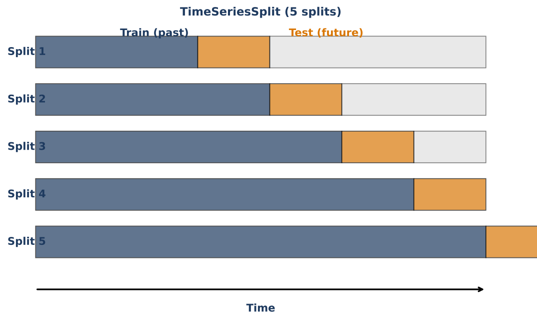


Finance critical: never let future data leak into training

TimeSeriesSplit

Train on the past, test on the future, roll forward

- `TimeSeriesSplit(n_splits=5)` creates 5 temporal folds
- `cross_val_score(pipe, X, y, cv=tscv)` scores each fold

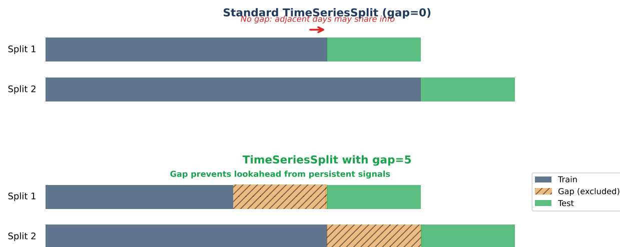


Each fold: train on earlier data, test on later data – respects the arrow of time

The Gap Parameter

Simulate real-world delay between prediction and action

- gap=0 (default): test starts right after training ends
- gap=5: skip 5 samples (one trading week) between sets
- Prevents autocorrelation from making predictions “too easy”

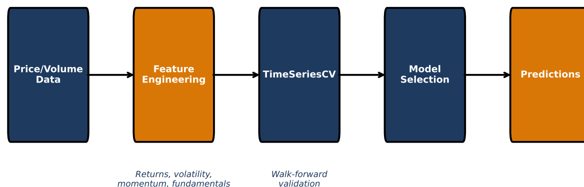


gap parameter matches CV to actual production deployment lag

Finance: Expanding vs Rolling Windows

Two strategies for financial time series

- **Expanding window**: train on all past data (grows over time)
- **Rolling window**: train on fixed-length recent window
- Rolling is better when market regimes shift (non-stationarity)

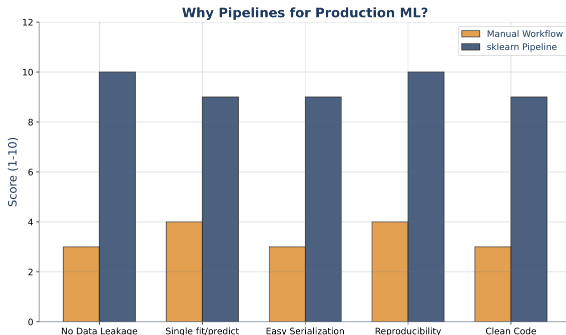


Expanding window vs rolling window – both valid for finance

Production Pipeline

One object captures the entire workflow

- `joblib.dump(pipe, 'model.joblib')` saves the pipeline
- `joblib.load('model.joblib')` restores it completely

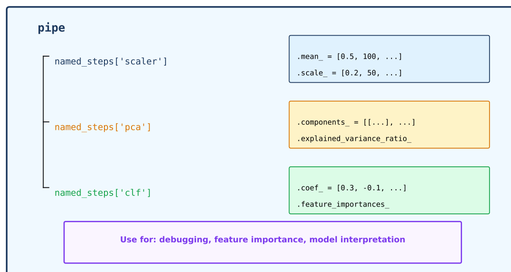


Single serialized object = reproducible, portable, deployable

Pipeline Inspection

Peek inside the assembly line

- `pipe.named_steps['scaler']` – access fitted transformer
- `pipe[:-1].transform(X)` – get intermediate output

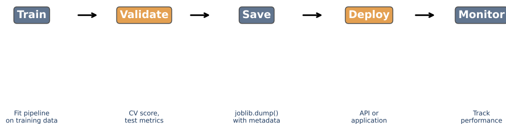


Inspect any step: `pipe.named_steps['pca'].explained_variance_ratio_`

Production Workflow

From development to deployment

- Train and validate → serialize with `joblib.dump()`
- Deploy: load and call `pipe.predict(new_data)`



Train → Save → Deploy → Load → Predict on new data

TimeSeriesSplit in Detail

Expanding window variant

- Default: training set grows each fold
- `max_train_size`: cap length for rolling window

TimeSeriesSplit expanding vs rolling – controlled by max_train_size

Complete ML Workflow

Seven stations on the assembly line

1. **Define columns:** separate numeric vs categorical features
2. **Build preprocessor:** ColumnTransformer with sub-pipelines
3. **Add model:** append estimator as final pipeline step
4. **Cross-validate:** estimate generalization (use TimeSeriesSplit for finance)
5. **Tune:** GridSearchCV or RandomizedSearchCV over the pipeline
6. **Refit:** train final pipeline on all training data
7. **Deploy:** serialize with joblib, load in production

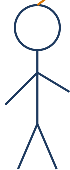
Key guarantee: every step inside the pipeline is refit on each training fold – **no data leakage**, ever.

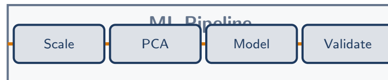
This 7-step recipe works for regression, classification, and clustering pipelines

Module 6 Complete!

Unsupervised Learning toolkit assembled

Module 6: Complete!





No data leakage allowed!

Next: Module 7 – Deep Learning

From raw data to deployed model – the assembly line is running

Hands-On Exercise (25 min)

Task: Build a Complete Prediction Pipeline

1. Create a pipeline: `StandardScaler` → `PCA(5)` → `Ridge`
2. Use `GridSearchCV` to tune `pca__n_components` ∈ [3, 5, 10] and `ridge__alpha` ∈ [0.1, 1, 10]
3. Evaluate with `TimeSeriesSplit(n_splits=5)`
4. Print best parameters and mean CV score
5. Save the best pipeline to disk with `joblib`

Extension: Add a `ColumnTransformer` for mixed numeric/categorical features and repeat.

Deliverable: Best params + CV score + saved model file.

Apply every concept from today: pipeline, CV, grid search, time series split

Summary

Module 6 Recap: Unsupervised Learning + Pipeline

- **L29 K-Means** – partition into K clusters (fast, scalable)
- **L30 Hierarchical** – dendrograms reveal nested structure
- **L31 PCA** – reduce dimensions, keep maximum variance
- **L32 Pipeline** – assemble preprocessing, model, and validation into one leak-free object

Key concepts from today:

- Pipeline prevents data leakage by construction
- Cross-validation gives robust performance estimates
- GridSearchCV / RandomizedSearchCV tune hyperparameters
- TimeSeriesSplit respects temporal order in finance

Next: Module 7 – Deep Learning, starting with the Perceptron (L33)

Pipeline = chain. GridSearchCV = try all. TimeSeriesSplit = respect time.