Intelligence Academy

Scikit Learn With Mathematics and Python

Lecture 02 Introduction: Why a unified API matters in Scikit Learn

Mejbah Ahammad

Introduction: Why a Unified API Matters—scikit-learn

Overview & Learning Goals

Learning goals. By the end of this lecture you should be able to (1) explain the *unified estimator API* in scikit-learn and the roles of `fit`/`transform`/`predict`/`score`; (2) compose leak-free workflows using `Pipeline` and `ColumnTransformer`; (3) run model selection with CV (`GridSearchCV`/`RandomizedSearchCV`) *inside* a pipeline; and (4) follow strict data conventions for $\mathbf{X}$ and y that make experiments reproducible and deployable.

What the "Unified API" Is

Definition. The unified API is a common *protocol* implemented by almost all sklearn objects:

`estimator.fit(X, y[, sample_weight])` $\rightarrow$ learn parameters, `.transform(X)` or `.pr`

Transformers (e.g., scalers, encoders) provide `fit+transform`; predictors (classifiers/regressors) provide `fit+predict` (and often `predict_proba` or `decision_function`). Because

every piece follows the same contract, components compose seamlessly into `Pipelines`, and search objects can tune *any* hyperparameters in that pipeline.

Key properties.

- **Stateless inputs, stateful objects:** `fit` reads data and writes learned state to attributes like `.coef_`, `.mean_`, `.categories_`.
- **Same method names across tasks:** You don't relearn APIs for each model.
- **Introspection:** `get_params`/`set_params` expose hyperparameters uniformly, enabling search.

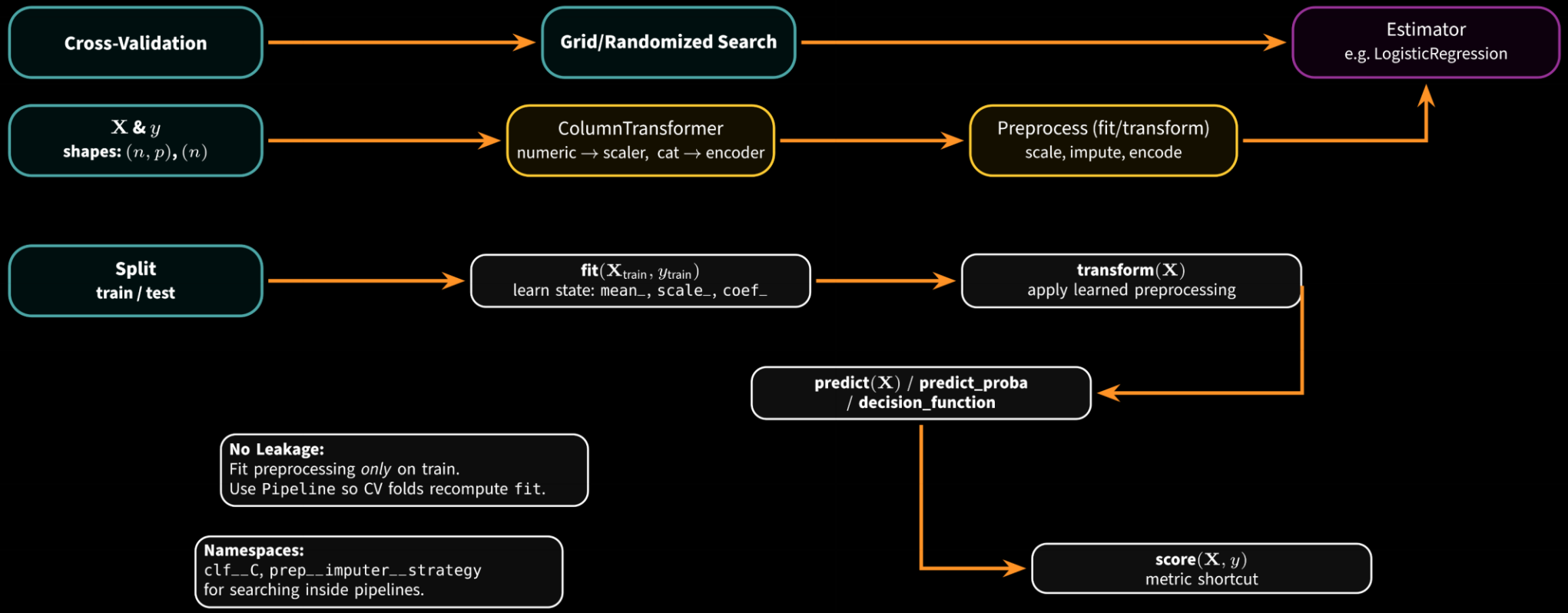
Why It Matters: Consistency, Composability, Reproducibility

Consistency. One mental model across preprocessing, modeling, and evaluation reduces bugs and switching costs. **Composability.** Any transformer → any estimator via `Pipeline`; mixed-type features via `ColumnTransformer`; unions via `FeatureUnion`. **Reproducibility.**

Explicit state, parameter namespaces (e.g., `clf__C`), and `random_state` enable faithful reruns. CV splits encapsulate evaluation so metrics aren't biased by leakage.

Core Data Conventions: $\mathbf{X}$, y, dtypes, shapes

Shapes. $\mathbf{X} \in \mathbb{R}^{n \times p}$ (rows=samples, columns=features); $y \in \mathbb{R}^n$ (regression) or $y \in \{0, \ldots, K-1\}^n$ (classification). Many estimators accept sparse $\mathbf{X}$ (CSR/CSC) for high-dimensional data. **Dtypes.** Numeric features typically `float64/float32`; categoricals are encoded via `OneHotEncoder` or similar; text via `TfidfVectorizer`. **Other conventions.** Keep `sample_weight` aligned with y; set `random_state` for stochastic steps; never call `fit` on test data (fit on train, transform/predict on test).



The Estimator Interface

Unified contract.

An *estimator* in scikit-learn follows a small, universal protocol:

$$\texttt{fit(X, y[, sample\_weight])} \quad \Rightarrow \quad \begin{cases} \texttt{transform(X)} & \text{(transformers)} \\ \texttt{predict(X)} & \text{(predictors)} \\ \texttt{score(X, y)} & \text{(quick metric)} \end{cases}$$

Many predictors also offer `predict_proba` (class probabilities) or `decision_function` (raw scores).

`fit`, `transform`, `predict`, `score`

fit. Reads data, learns state (e.g., `mean_`, `scale_`, `coef_`, `classes_`), and returns `self`. **transform.** Applies learned state to new $\mathbf{X}$ (scaling, encoding, projections). **predict.** Produces outputs: class labels (classification), real values (regression), cluster IDs (clustering). **score.** Con-

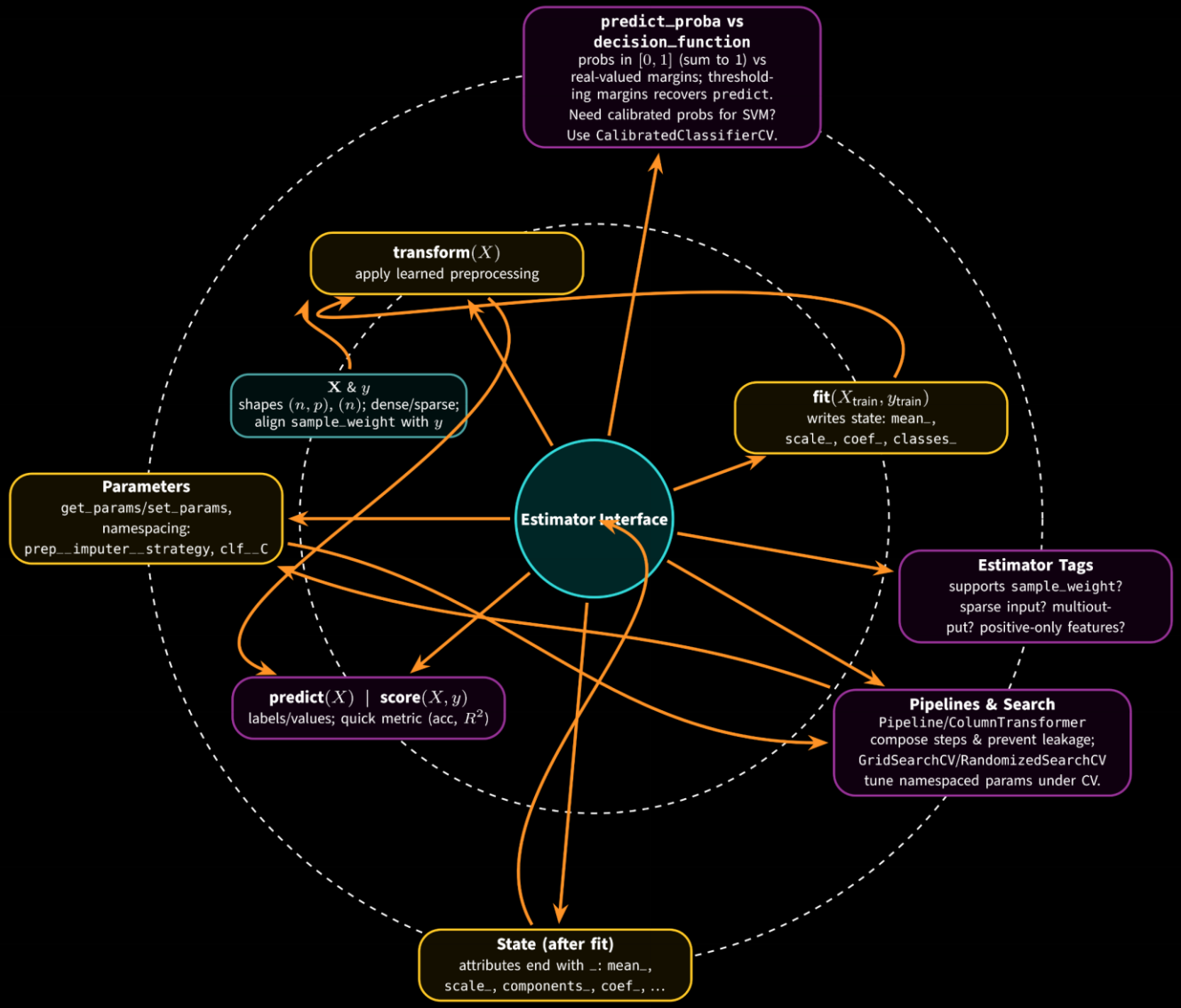
venience metric (e.g., accuracy for classifiers, R^2 for regressors); prefer explicit scorers inside CV.

`predict_proba` vs. `decision_function`

`predict_proba` returns calibrated probabilities in $[0, 1]$ that sum to `1` across classes (when supported: logistic regression, naive Bayes, tree ensembles, etc.). **`decision_function`** returns real-valued margins/scores (e.g., signed distance to SVM hyperplane). Thresholding these scores yields the same labels as `predict`; they are not probabilities unless calibrated.

Parameters & State: `get_params`/`set_params`, Estimator Tags

Parameters are constructor arguments (hyperparameters) accessible via `get_params`/`set_params`; search objects (`GridSearchCV`/`RandomizedSearchCV`) use them uniformly, including inside `Pipelines` via namespacing (e.g., `clf__C`). **State** are learned attributes created during `fit` (names end with `_`). **Estimator tags** (internal metadata) communicate capabilities (supports `sample_weight`? accepts sparse? multioutput?), helping utilities choose safe defaults.



Pipelines & No-Leakage Patterns

Why this matters.

Any preprocessing that "sees" the test fold leaks information and inflates metrics. Wrap *every* step—imputation, encoding, scaling, feature selection, the estimator—inside a single `Pipeline`. Tune only via CV objects that refit the *entire* pipeline per split.

`Pipeline` Essentials

What it gives you.

- **Single contract:** `pipe.fit(X_train, y_train)` learns state for all steps; `pipe.predict(X_test)` applies the same transforms before the model.
- **Leakage-proof CV:** `GridSearchCV`/`RandomizedSearchCV` refit the whole pipeline inside each fold.
- **Namespacing:** hyperparameters are addressed as `step__param` (e.g., `prep__num__scaler__with_mean`, `clf__C`).

- **Reproducibility:** set `random_state` on stochastic steps; persist the *pipeline* (joblib) so production transforms match training.
- **Performance:** enable `Pipeline(memory=...)` caching when heavy transformers are upstream of tuning.

Minimal recipe (illustrative).

```
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV, train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, \
    random_state=0)

pipe = Pipeline([
    ("prep",   ... your ColumnTransformer ...),
    ("clf",    ... your estimator ...)
])

param_grid = {
    "clf__C": [0.1, 1, 10]
}
cv = GridSearchCV(pipe, param_grid=param_grid, scoring="accuracy", n_jobs=-1, \
    cv=5)
cv.fit(X_train, y_train)
cv.score(X_test, y_test)
```

`ColumnTransformer` for Mixed Features

Principle.

Route different column subsets through tailored sub-pipelines:

- **Numeric** $\rightarrow$ impute $\rightarrow$ scale.
- **Categorical** $\rightarrow$ impute $\rightarrow$ one-hot encode.
- **Passthrough** already clean features (e.g., engineered flags).

Practical tips.

- Use column names or indices; `make_column_selector(dtype_include=...)` helps auto-select.
- For unseen categories in validation/test, set `OneHotEncoder(handle_unknown="ignore")`.
- Control leftovers via `remainder="drop"` or `"passthrough"`.

- Be aware of sparsity: one-hot often returns sparse matrices; ensure the estimator supports sparse input.

Preprocessing: Scaling, Encoding, Imputation

Numeric transforms.

- **Imputation first:** `SimpleImputer(strategy="median")` is robust to outliers; consider `KNNImputer` when values are informative across features.
- **Scaling next:**
  - `StandardScaler` for models assuming standardized features (linear/SVM, kNN).
  - `MinMaxScaler` for bounded features or distance models with range sensitivity.
  - `RobustScaler` when heavy tails/outliers are prominent.

Categorical transforms.

- **Impute:** `SimpleImputer(strategy="most_frequent")` (or a sentinel like `constant`).

- **Encode:** `OneHotEncoder(drop="if_binary", handle_unknown="ignore")` for unordered categories.
- **OrdinalEncoder** only for truly ordered categories (beware of injecting fake order).

Composition pattern.

- Build per-type `Pipelines` (numeric, categorical) $\rightarrow$ plug into `ColumnTransformer` named `"prep"` $\rightarrow$ wrap with final estimator in top-level `Pipeline`.

Train/Test Discipline: Fit vs. Transform

Non-negotiables.

- **Split first, fit later:** perform `train_test_split` (or CV) before any `fit`. Do *not* prefit scalers/encoders on all data.
- **Train-only state:** `fit` on train folds; use `transform`/`predict` on validation/test folds.

- **Tune with CV**: call `GridSearchCV`/`RandomizedSearchCV` on the *pipeline*; never tune components outside it.
- **Time series/groups:** use `TimeSeriesSplit` for temporal data; `GroupKFold`/`GroupShuffleSplit` when groups must not cross folds.
- **Feature selection/leakage:** feature selection and target-aware transforms *must* live inside the pipeline so they refit per fold.

Quick checklist.

- ☐ All preprocessing is inside `Pipeline` / `ColumnTransformer`.
- ☐ Hyperparameters addressed via namespacing (`prep__*`, `clf__*`).
- ☐ CV uses appropriate splitter (stratified, grouped, or temporal).
- ☐ `random_state` set on stochastic steps.
- ☐ Persist the final *pipeline* for deployment (not just the estimator).

Model Selection & Validation

Goal.

Estimate *generalization* honestly and pick hyperparameters that maximize a chosen metric, while avoiding leakage and optimistic bias.

Cross-Validation: `KFold`, `StratifiedKFold`, `GroupKFold`, `TimeSeriesSplit`

- **KFold** (iid features/labels): uniform folds (shuffle optional).
- **StratifiedKFold** (classification): preserves class proportions per fold.
- **GroupKFold** / **GroupShuffleSplit**: keep same group (e.g., user, subject, scene) from crossing train/val.
- **TimeSeriesSplit**: expanding window; validation strictly in the future w.r.t. training to respect causality.
- Choose the *splitter* that matches the data-generation process; use `Pipeline` so all fitting happens only on the training portion of each fold.

Scoring Metrics (Classification, Regression, Ranking)

- **Classification:** `accuracy`, `balanced_accuracy`, `f1`, `precision/recall`, `roc_auc`, `average_precision`.
- **Regression:** `r2`, `neg_mean_absolute_error`, `neg_mean_squared_error`, `neg_median_absolute_error`.
- **Ranking/Probabilities:** `roc_auc`, `average_precision`, Brier score (`neg_brier_score`). For imbalanced data, prefer threshold-free metrics (ROC-AUC/PR-AUC).
- Set `scoring=` explicitly; don't rely on defaults unless you really want them (e.g., `score=` accuracy or R^2).

Hyperparameter Search: `GridSearchCV`, `RandomizedSearchCV`, Successive Halving

- **GridSearchCV:** exhaustive on small, structured grids.

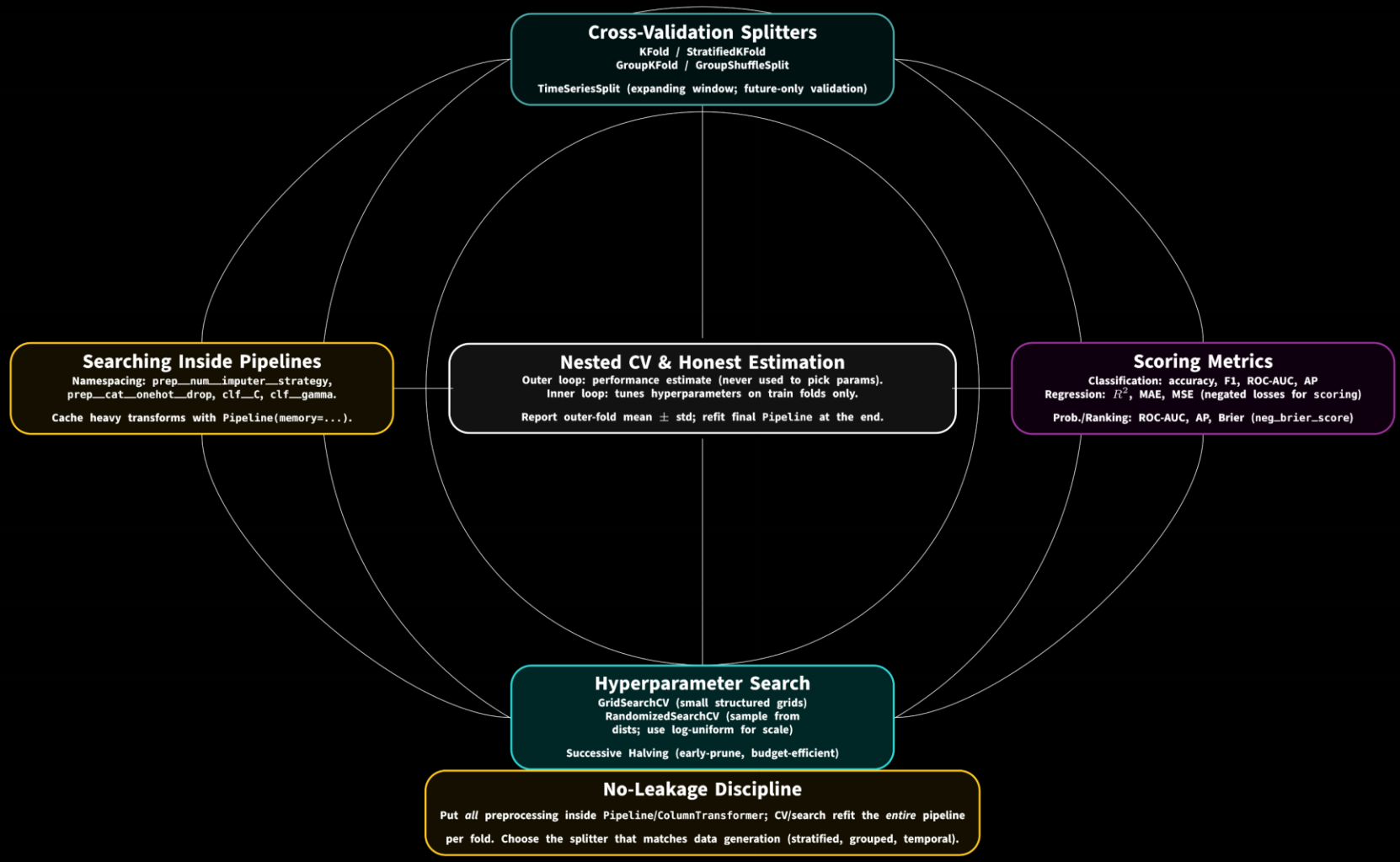
- **RandomizedSearchCV:** sample from distributions; far better when the space is wide or unbounded (log-uniform for C, learning rates).
- **Successive Halving** (`HalvingGridSearchCV`/`HalvingRandomSearchCV`): allocate more budget to promising configs using early-stopping style pruning.
- Always search *over a full `Pipeline`* to keep preprocessing and estimator tied together.

Searching Inside Pipelines (Param-Grid Namespacing)

- Use `step__param` keys, e.g., `prep__num__imputer__strategy`, `prep__cat__onehot__drop`, `clf__C`.
- Prefer **log-spaced** ranges for scale parameters (e.g., C, α, γ).
- Cache heavy transformers with `Pipeline(memory=...)` when tuning to save time.

Nested CV & Honest Performance Estimation

- **Nested CV** = outer CV for performance estimation, inner CV for hyperparameter tuning. Prevents the “validation set became tuning set” bias.
- Report the *outer* CV score distribution (mean $\pm$ std); refit on the full data only after model choice is fixed.
- For groups or time series, use `GroupKFold` or `TimeSeriesSplit` in *both* inner and outer loops.



Problem Types & Interfaces

Unifying idea.

Regardless of task (classification, regression, clustering, dimensionality reduction), estimators expose the same core API: `fit(X, y?)` → learn state; `predict(X)` → outputs; optional `transform(X)` or `predict_proba(X)` when defined. Compose everything in a `Pipeline`.

Classification: Binary, Multiclass, Multilabel

Targets and shapes.

- **Binary** and **multiclass:** $y \in \{0, \ldots, K-1\}$ with shape $(n,)$.
- **Multilabel-indicator:** $Y \in \{0,1\}^{n \times L}$ (L=#labels); `scipy.sparse` CSR often efficient.
- **Multioutput (vector targets):** $Y \in \mathbb{R}^{n \times d}$ for multiple dependent variables.

Interfaces.

- `ClassifierMixin`: `fit`, `predict`, `predict_proba` or `decision_function`, `score`.

- **Problem wrappers:**
  - `OneVsRestClassifier` (multilabel / multiclass), `OneVsOneClassifier` (small K).
  - `OutputCodeClassifier` (error-correcting codes for multiclass).
  - `MultiOutputClassifier` for independent per-output models.

Common choices.

- Linear models: `LogisticRegression` (strong baseline; supports class weights).
- Margin models: `LinearSVC`/`SVC` (use `probability=True` to enable calibrated `predict_proba` via Platt).
- Tree ensembles: `RandomForestClassifier`, `GradientBoostingClassifier`, `HistGradientBoostingClassifier`.
- Naive Bayes for sparse text: `MultinomialNB`, `ComplementNB`.

Scoring (`scoring=`):

`accuracy`, `balanced_accuracy`, `f1`, `f1_macro`, `roc_auc` (OVR/OVO), `average_precision` (PR-AUC). Prefer threshold-free metrics (ROC/PR) when classes are imbalanced.

Minimal pattern.

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression

prep = ColumnTransformer([
  ("num", Pipeline([("imp", SimpleImputer(strategy="median")),
                    ("sc",  StandardScaler())]), num_cols),
  ("cat", Pipeline([("imp", SimpleImputer(strategy="most_frequent")),
                    ("oh",  OneHotEncoder(handle_unknown="ignore"))]), cat_cols)
])

pipe = Pipeline([("prep", prep), ("clf", LogisticRegression(max_iter=1000))])
```

Regression & `TransformedTargetRegressor`

Targets and shapes.

- **Single-output:** $y \in \mathbb{R}^n$.
- **Multioutput:** $Y \in \mathbb{R}^{n \times d}$ with `MultiOutputRegressor` wrapping any base regressor.

Interfaces.

`RegressorMixin`: `fit`, `predict`, `score` (default R^2).

Common choices.

- Linear: `Ridge`, `Lasso`, `ElasticNet`.
- Tree ensembles: `RandomForestRegressor`, `HistGradientBoostingRegressor`.
- Count/positive targets: `PoissonRegressor`, `TweedieRegressor` (power defines distribution).

`TransformedTargetRegressor` (TTR).

Wraps a regressor and applies a reversible transform to y (e.g., log), fitting in transformed space and inverting on prediction.

```python
from sklearn.compose import TransformedTargetRegressor
from sklearn.preprocessing import FunctionTransformer
import numpy as np

log = FunctionTransformer(np.log1p, inverse_func=np.expm1, check_inverse=False)
ttr = TransformedTargetRegressor(regressor=Ridge(), transformer=log)
ttr.fit(X_train, y_train)  # fits Ridge on log1p(y)
y_pred = ttr.predict(X_test)  # returns on original scale
```

Scoring (`scoring=`):

`r2`, `neg_mean_absolute_error`, `neg_mean_squared_error`, `neg_median_absolute_error`. For skewed targets, prefer MAE or use TTR.

Unsupervised: Clustering & Dimensionality Reduction

Clustering.

- `KMeans`/`MiniBatchKMeans`: centroid-based; set `n_clusters`; `fit` → `labels_`, `cluster_centers_`; `predict` assigns clusters.
- `DBSCAN`/`HDBSCAN*`: density-based; automatic #clusters; handles noise; `fit`→`labels_` (-1=noise). (*HDBSCAN is external.)
- `AgglomerativeClustering`: hierarchical with linkage choices.
- `SpectralClustering`: graph-based; useful for non-convex structures.

Dimensionality reduction (transformers).

- `PCA` (dense), `TruncatedSVD` (sparse): `fit` → `components_`; `transform` gives low-dim embeddings; use in `Pipeline`.
- `KernelPCA`, `Isomap`, `LocallyLinearEmbedding`, `TSNE` (visualization; not a transformer in the strict sense for generalization).

Internal validation.

Silhouette score, Davies–Bouldin, Calinski–Harabasz (require only X and labels). For DR, inspect explained variance (`PCA.explained_variance_ratio_`).

Probabilistic Outputs & Calibration

Two interfaces.

- `predict_proba(X)` $\in [0, 1]$ per class, rows sum to 1 (if supported).
- `decision_function(X)`: real-valued margins/scores; thresholding yields labels; not calibrated.

Calibration.

Use `CalibratedClassifierCV` to convert margins to probabilities via `method="sigmoid"` (Platt) or `"isotonic"` (flexible for more data).

```
from sklearn.calibration import CalibratedClassifierCV
base = LinearSVC()  # margins only
clf = CalibratedClassifierCV(base, method="sigmoid", cv=5)
clf.fit(X_train, y_train)            # now has predict_proba
```

When to care.

- Decision-theory / cost-sensitive thresholds, ranking, active learning.
- Evaluate with Brier score (`neg_brier_score`) and calibration curves; ROC-AUC/PR-AUC stay threshold-free.

Imbalanced Data: `class_weight`, sampling

Built-in weighting.

- Set `class_weight="balanced"` (e.g., Logistic Regression, SVM, trees) to weight inversely to class frequency.
- Or pass `sample_weight=` in `fit(X, y, sample_weight=w)` (must align with y).

Sampling strategies.

- *Under/over-sampling* can be effective; if you use external samplers (e.g., imbalanced-learn), integrate them inside a *Pipeline* and apply within CV folds to avoid leakage.
- Prefer threshold-free metrics (ROC-AUC/PR-AUC); report per-class metrics or macro-averages.

Baseline recipe (safe).

```python
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import StratifiedKFold, cross_val_score

clf = LogisticRegression(class_weight="balanced", max_iter=1000)
cv  = StratifiedKFold(n_splits=5, shuffle=True, random_state=0)
scores = cross_val_score(clf, X, y, scoring="roc_auc", cv=cv, n_jobs=-1)
print(scores.mean(), scores.std())
```

Checklist.

- ☐ Choose target shape correctly: $(n,)$ vs. (n, L) vs. multioutput.

- ☐ Keep all preprocessing in a `Pipeline`/`ColumnTransformer`.
- ☐ Pick a splitter matching the data: stratified, grouped, or temporal.
- ☐ Set `scoring=` explicitly to match the task and class balance.
- ☐ Calibrate if downstream decisions need probabilities.
- ☐ For imbalanced data, use `class_weight` or (carefully) sampling inside CV.

Interoperability & Extensibility

Big idea.

scikit-learn speaks "array-like": `numpy.ndarray`, `pandas.DataFrame`, and `scipy.sparse`. The unified API (`fit`/`transform`/`predict`) lets you plug custom pieces, merge feature branches, and persist full pipelines for reproducible inference.

NumPy/pandas/SciPy: Dense vs. Sparse

Accepted inputs.

- **NumPy dense** (`float32/float64/int`): fastest path for numeric features.
- **pandas** (`DataFrame/Series`): column names, dtypes, missing values; many transformers now support `set_output(transform="pandas")` to return DataFrames.
- **SciPy sparse** (`csr_matrix/csc_matrix`): essential for high-dimensional one-hot/text. Most linear models/trees accept CSR/CSC.

Dense vs. sparse tips.

- **One-hot ⇒ sparse** by default. Keep it sparse to save memory; use `.toarray()` only if the estimator requires dense.
- **CSR vs. CSC:** CSR is best for row-wise ops (mini-batches); CSC can be better for column-wise solvers. Most estimators accept both.
- **Dtypes & NA:** scale numeric to `float64` by default; impute before scaling. For pandas categories, use `OneHotEncoder(handle_unknown="ignore")`.

Custom Transformers/Estimators (`BaseEstimator, *Mixin`)

Why roll your own?

Encode domain logic, feature engineering, learned encoders, or model wrappers—while staying compatible with `Pipeline`, `GridSearchCV`, and `ColumnTransformer`.

Transformer skeleton (safe for CV).

```python
from sklearn.base import BaseEstimator, TransformerMixin
import numpy as np

class RatioAndLog(BaseEstimator, TransformerMixin):
    def __init__(self, a_col, b_col, add_log=True):
        self.a_col = a_col
        self.b_col = b_col
        self.add_log = add_log

    def fit(self, X, y=None):
        # learn nothing, but could learn statistics here
        return self

    def transform(self, X):
```

```
        A, B = X[:, self.a_col], X[:, self.b_col]
        ratio = np.divide(A, np.maximum(B, 1e-12))
        if self.add_log:
            return np.c_[X, ratio, np.log1p(np.abs(ratio))]
        return np.c_[X, ratio]
```

Estimator skeleton (with state & params).

```
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.linear_model import LogisticRegression

class ThresholdedLR(BaseEstimator, ClassifierMixin):
    def __init__(self, C=1.0, threshold=0.5):
        self.C = C
        self.threshold = threshold
        self._lr = LogisticRegression(C=C, max_iter=1000)

    def fit(self, X, y):
        self._lr.set_params(C=self.C)
        self._lr.fit(X, y)
        return self
```

```
14
15      def predict(self, X):
16          proba = self._lr.predict_proba(X)[:, 1]
17          return (proba >= self.threshold).astype(int)
18
19      def predict_proba(self, X):
20          return self._lr.predict_proba(X)
```

Mixins.

`TransformerMixin` supplies a default `fit_transform`; `ClassifierMixin`/`RegressorMixin` supply `score`. Inherit from `BaseEstimator` to get `get_params`/`set_params` (required for grid search).

Feature Unions (`FeatureUnion`/`make_union`)

Parallel feature branches.

Run independent pipelines in parallel and horizontally concatenate outputs. Great for mixing text, counts, statistics, embeddings, image descriptors, etc.

```python
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.feature_selection import SelectKBest
from sklearn.decomposition import TruncatedSVD
from sklearn.pipeline import FeatureUnion, make_union
from sklearn.linear_model import LogisticRegression

num_pipe = make_pipeline(SimpleImputer(strategy="median"), StandardScaler())
cat_pipe = make_pipeline(SimpleImputer(strategy="most_frequent"),
                         OneHotEncoder(handle_unknown="ignore"))
prep = ColumnTransformer([("num", num_pipe, num_cols),
                          ("cat", cat_pipe, cat_cols)])

branch_stats = make_pipeline(SelectKBest(k=50))           # numeric stats
branch_svd   = make_pipeline(TruncatedSVD(n_components=50))  # dim-red

union = make_union(branch_stats, branch_svd)

```

```
pipe = Pipeline([
    ("prep",  prep),       # column-wise preprocessing
    ("union", union),      # parallel branches concatenated
    ("clf",   LogisticRegression(max_iter=1000))
])
```

Notes.

Each branch must return the same number of rows; sparse outputs are concatenated efficiently. Namespacing applies (e.g., `union__pipeline-1__truncatedsvd__n_components`).

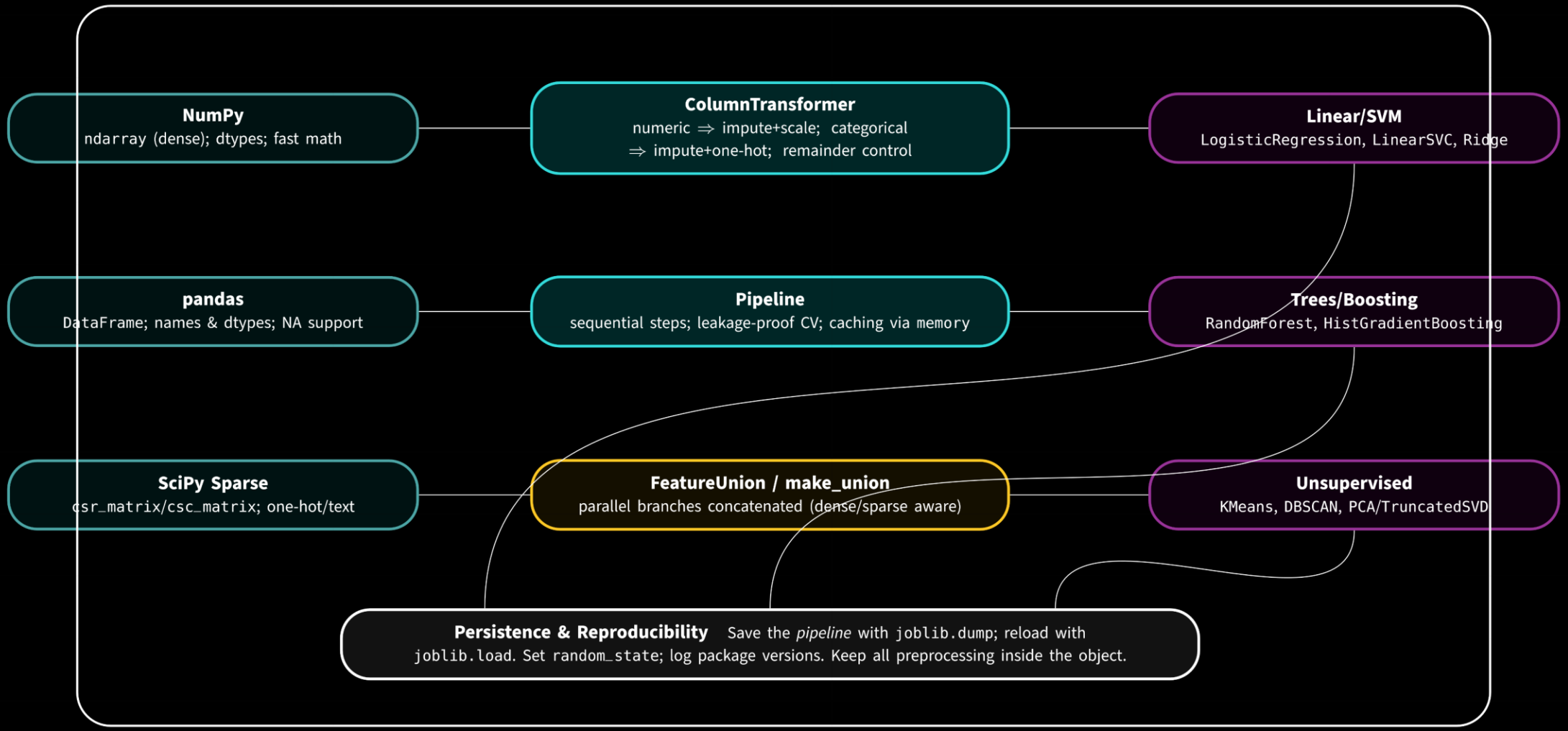
Persistence with `joblib` & Reproducibility

Persist the pipeline*, not just the model.*

```
import joblib
joblib.dump(pipe, "model_pipeline.joblib")
pipe2 = joblib.load("model_pipeline.joblib")  # ready to .predict(X_new)
```

Reproducibility checklist.

- Set `random_state` for stochastic steps (splitters, models, imputers like KNN).
- Log library versions (`sklearn`, `numpy`, `scipy`, `pandas`).
- Keep all transforms inside the persisted object (avoid ad-hoc preprocessing outside).
- For heavy tuning, enable `Pipeline(memory=...)` caching to speed up repeated `fit/transform` in CV.



Performance & Ops Considerations

Goal.

Fast, reliable ML hinges on three levers: *parallelism*, *memory efficiency*, and *throughput*. Wrap it with reproducible deployment (versions, threads, CPU features) and you're production-ready.

Parallelism with `n_jobs`

Where it works.

Many estimators and utilities accept `n_jobs` for parallel CPU use:

- Model families: trees/forests (`RandomForest*`, `HistGradientBoosting*`), neighbors (`NearestNeighbors`, `KNN*`), linear models with `n_jobs` in some solvers, decomposition (PCA randomized SVD), clustering (`KMeans` with `n_init>1` in newer versions uses parallel inits).

- Model selection: `GridSearchCV`, `RandomizedSearchCV`, `Halving*`, `cross_val_score`.

Threading vs. processes.

scikit-learn uses **joblib** backends; by default, threads are used (good for NumPy/BLAS). Be careful with *nested* parallelism: e.g., `GridSearchCV(n_jobs=-1)` over a `RandomForest(n_jobs=-1)` multiplies cores. Typically:

$$\text{outer } n\_jobs > 1, \quad \text{inner } n\_jobs = 1.$$

BLAS/OpenMP caps.

Linear algebra may spawn threads independently. Cap them to avoid oversubscription:

```
# before importing numpy/scikit-learn
import os
os.environ["OMP_NUM_THREADS"] = "1"
os.environ["MKL_NUM_THREADS"] = "1"
```

Memory/Efficiency: Sparse, Incremental (`partial_fit`)

Sparse saves RAM.

Keep high-cardinality one-hot/text as `scipy.sparse` (`csr_matrix`/`csc_matrix`). Many estimators accept sparse directly; avoid `.toarray()` unless required.

Dtypes.

Prefer `float32` for very large matrices if supported by the estimator (halves memory). Imputation/scaling typically promote to float.

Incremental learning.

For streams or huge datasets, use estimators that support `partial_fit`: `SGDClassifier/Regressor`, `Perceptron`, `PassiveAggressive*`, `MiniBatchKMeans`, `MiniBatchDictionaryLearning`, `BernoulliNB/MultinomialNB` (some), `CalibratedClassifierCV` (on top of margin models after fit).

```
clf = SGDClassifier(loss="log_loss", random_state=0)
for Xb, yb in stream(batches):
    clf.partial_fit(Xb, yb, classes=[0,1])  # classes once
```

Pipeline caching.

When tuning, reuse deterministic steps:

```
from tempfile import mkdtemp
from sklearn.pipeline import Pipeline
cachedir = mkdtemp()
pipe = Pipeline([...], memory=cachedir)   # transformers with same params are \
    cached
```

Throughput: Vectorization, Batching, Warm Starts

Vectorize first.

Favor NumPy ops over Python loops inside custom transformers. Use `ColumnTransformer` to push work into compiled transformers.

Batching for predict.

For very large X, call `predict` in chunks to bound memory:

```python
def batched_predict(model, X, batch=100_000):
    out = []
    for i in range(0, X.shape[0], batch):
        out.append(model.predict(X[i:i+batch]))
    return np.concatenate(out)
```

Warm starts.

Resume training with previous state when supported (`warm_start=True`):

- `RandomForest*` (grow more trees), `GradientBoosting*`, some linear models (e.g., `LogisticRegression` with `solver="saga"`), `KMeans` (pass `init`).

For targets with strong skew/heteroscedasticity, `TransformedTargetRegressor` accelerates convergence on a better-behaved scale.

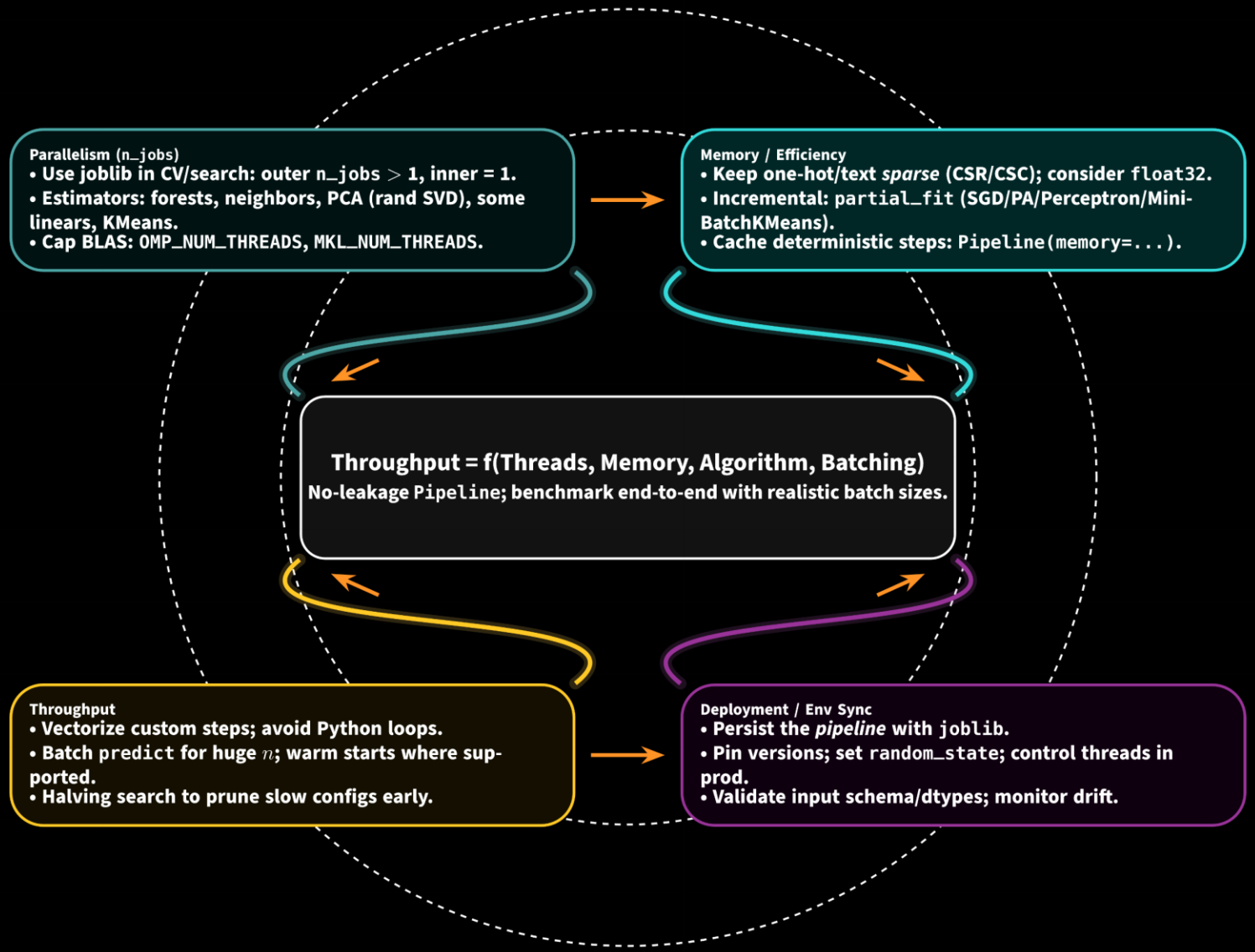
Deployment Notes & Environment Sync

Persist the whole pipeline.

```
import joblib
joblib.dump(pipe, "pipeline.joblib")
pipe = joblib.load("pipeline.joblib")          # ready to predict
```

Pin the environment.

- Record versions: `sklearn`, `numpy`, `scipy`, `pandas`, `joblib`.
- Fix `random_state` everywhere for determinism across re-runs.
- Align thread caps in prod: `OMP_NUM_THREADS`, `MKL_NUM_THREADS`; avoid nested parallelism.
- CPU features: models persisted with one BLAS can load on another, but throughput may differ—benchmark.
- Security: validate inputs (schema/dtypes), sanitize paths if loading untrusted artifacts.



Case Studies & Demos

Goal.

Four end-to-end mini-workflows that showcase the unified API across tabular, regression with target transforms, text, and time-series — all leakage-safe, CV-correct, and reproducible.

Tabular Classification: Full Pipeline + CV

What you'll see.

Mixed numeric/categorical features, column-wise preprocessing inside a `Pipeline`, and stratified cross-validation with explicit scoring.

Pattern (safe baseline).

```
from sklearn.model_selection import StratifiedKFold, cross_val_score
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
```

```
from sklearn.linear_model import LogisticRegression

num_pipe = make_pipeline(SimpleImputer(strategy="median"),
                         StandardScaler())
cat_pipe = make_pipeline(SimpleImputer(strategy="most_frequent"),
                         OneHotEncoder(handle_unknown="ignore"))

prep = ColumnTransformer([
    ("num", num_pipe, num_cols),
    ("cat", cat_pipe, cat_cols)
])

pipe = Pipeline([
    ("prep", prep),
    ("clf", LogisticRegression(max_iter=1000, class_weight="balanced"))
])

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=0)
scores = cross_val_score(pipe, X, y, scoring="roc_auc", cv=cv, n_jobs=-1)
print(scores.mean(), scores.std())
```

Notes.

`class_weight="balanced"` helps when classes are skewed; choose threshold-free metrics (`roc_auc`/PR-AUC) for selection.

Regression with Target Transformation & Residuals

Why transform targets?

Skewed/positive-only targets often fit better on a log-like scale. `TransformedTargetRegressor` (TTR) handles invertible transforms cleanly.

Pattern.

```
import numpy as np
from sklearn.compose import TransformedTargetRegressor
from sklearn.preprocessing import FunctionTransformer
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import KFold, cross_val_predict

```

```
log = FunctionTransformer(np.log1p, inverse_func=np.expm1, check_inverse=False)
ttr = TransformedTargetRegressor(regressor=Ridge(alpha=1.0), transformer=log)

kf = KFold(n_splits=5, shuffle=True, random_state=0)
y_pred = cross_val_predict(ttr, X, y, cv=kf, n_jobs=-1)
print("MAE:", mean_absolute_error(y, y_pred))
```

Residuals check.

After selecting the model, refit on full training data and plot residuals vs. predictions to probe heteroscedasticity/outliers.

Text Workflow: `TfidfVectorizer` + Linear Models

Why linear on TF-IDF?

Sparse high-dimensional text often shines with linear margin models.

Pattern (binary/multiclass).

```python
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import SGDClassifier   # 'log_loss' = logistic
from sklearn.model_selection import StratifiedKFold, cross_val_score

pipe = Pipeline([
    ("tfidf", TfidfVectorizer(min_df=3, ngram_range=(1,2))),
    ("clf",   SGDClassifier(loss="log_loss", alpha=1e-4,
                            max_iter=1000, tol=1e-3, random_state=0))
])

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=0)
scores = cross_val_score(pipe, texts, labels, scoring="f1_macro", cv=cv, \
     n_jobs=-1)
print(scores.mean(), scores.std())
```

Tips.

Use `class_weight` or threshold-free metrics for imbalance; `CalibratedClassifierCV` if you need calibrated probabilities.

Time-Series Splits & Leakage Caveats

Golden rule.

Never learn anything from the future. Use `TimeSeriesSplit` (expanding window) and keep all preprocessing inside the pipeline.

Pattern.

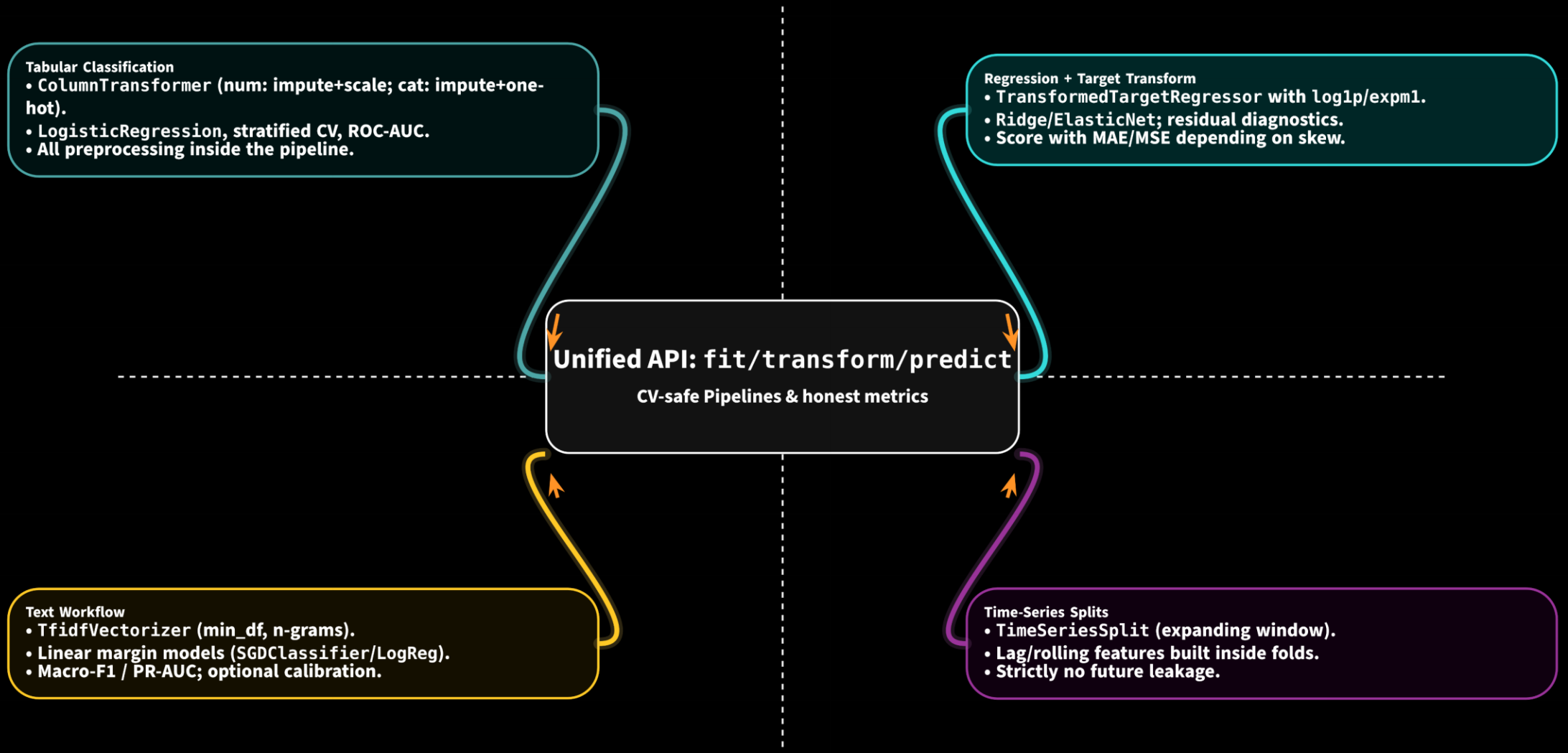
```python
from sklearn.model_selection import TimeSeriesSplit, cross_val_score
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Ridge

pipe = Pipeline([
    ("sc",   StandardScaler(with_mean=False)),  # safe for sparse if any
    ("reg",  Ridge(alpha=1.0))
])

tscv = TimeSeriesSplit(n_splits=5)
```

```
scores = cross_val_score(pipe, X_ts, y_ts, cv=tscv, \
     scoring="neg_mean_absolute_error")
print(-scores.mean(), scores.std())
```

Leakage watchlist.

Lag features? Build them *inside* CV folds (fit on past, apply to hold-out). Rolling stats must exclude current/forward windows. If groups exist (stores, users), combine grouped splits with temporal logic.

Intelligence Academy



Lab & Problem Set

Format.

Each lab is end-to-end and leakage-safe. Submit (i) *code*, (ii) *CV results* (mean±std with chosen metric), and (iii) a *one-paragraph reflection* on design and tradeoffs.

Build an End-to-End Mixed-Feature Pipeline

Task.

Given a tabular dataset with numeric & categorical columns:

- Build a `ColumnTransformer`: numeric → `SimpleImputer(median)` + `StandardScaler`; categorical → `SimpleImputer(most_frequent)` + `OneHotEncoder(handle_unknown="ignore")`.
- Wrap in `Pipeline` with a classifier (`LogisticRegression` or `HistGradientBoostingClassifier`).
- Evaluate with `StratifiedKFold` (5-fold), scoring with ROC–AUC & F1.

Starter.

```python
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import StratifiedKFold, cross_validate

num_pipe = make_pipeline(SimpleImputer(strategy="median"), StandardScaler())
cat_pipe = make_pipeline(SimpleImputer(strategy="most_frequent"),
                         OneHotEncoder(handle_unknown="ignore"))
prep = ColumnTransformer([("num", num_pipe, num_cols),
                          ("cat", cat_pipe, cat_cols)])

pipe = Pipeline([("prep", prep),
                 ("clf", LogisticRegression(max_iter=1000, \
                     class_weight="balanced"))])

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=0)
scores = cross_validate(pipe, X, y, scoring=["roc_auc","f1"], cv=cv, n_jobs=-1)
print("ROC-AUC:", scores["test_roc_auc"].mean(), scores["test_roc_auc"].std())
```

```
print("F1:",     scores["test_f1"].mean(),     scores["test_f1"].std())
```

Hyperparameter Search with a Custom Scorer

Task.

Define a **cost-sensitive** scorer that weights FN higher than FP. Run `GridSearchCV` on your pipeline; report best params and CV score.

Starter.

```
import numpy as np
from sklearn.metrics import make_scorer

def cost_fn(y_true, y_pred, w_fn=5.0, w_fp=1.0):
    y_true = np.asarray(y_true); y_pred = np.asarray(y_pred)
    fn = ((y_true==1) & (y_pred==0)).sum()
    fp = ((y_true==0) & (y_pred==1)).sum()
    return -(w_fn*fn + w_fp*fp)  # higher is better -> negative cost

cost_scorer = make_scorer(cost_fn, greater_is_better=True)
```

```
11
12  from sklearn.model_selection import GridSearchCV
13  param_grid = {
14    "clf__C": np.logspace(-2, 2, 7)
15  }
16  gcv = GridSearchCV(pipe, param_grid=param_grid, scoring=cost_scorer,
17                     cv=cv, n_jobs=-1, refit=True)
18  gcv.fit(X, y)
19  print("Best params:", gcv.best_params_)
20  print("Best score:", gcv.best_score_)
```

Note.

Use namespacing (`clf__C`) to reach inside the pipeline.

Implement a Custom Transformer

Task.

Create `OutlierClipper` that clips numeric features to a multiple of the IQR, learned on training folds only. Integrate into `num_pipe` before scaling.

Starter.

```python
from sklearn.base import BaseEstimator, TransformerMixin
import numpy as np

class OutlierClipper(BaseEstimator, TransformerMixin):
    def __init__(self, k=3.0):
        self.k = k

    def fit(self, X, y=None):
        Q1 = np.percentile(X, 25, axis=0)
        Q3 = np.percentile(X, 75, axis=0)
        self.q1_ = Q1; self.q3_ = Q3
        self.iqr_ = np.maximum(Q3 - Q1, 1e-12)
        return self

    def transform(self, X):
        lo = self.q1_ - self.k * self.iqr_
        hi = self.q3_ + self.k * self.iqr_
        return np.clip(X, lo, hi)

```

```
# insert into numeric pipeline
num_pipe = make_pipeline(OutlierClipper(k=2.5),
                         SimpleImputer(strategy="median"),
                         StandardScaler())
```

Checks.

Verify `get_params` / `set_params` work, so it's tunable in grid search (`num__outlierclipper__k`).

Compare CV Strategies & Run a Leakage Check

Task.

Compare `KFold`/`StratifiedKFold`/`GroupKFold`/`TimeSeriesSplit` on a dataset with known groups or time order. Explain which is honest for your data.

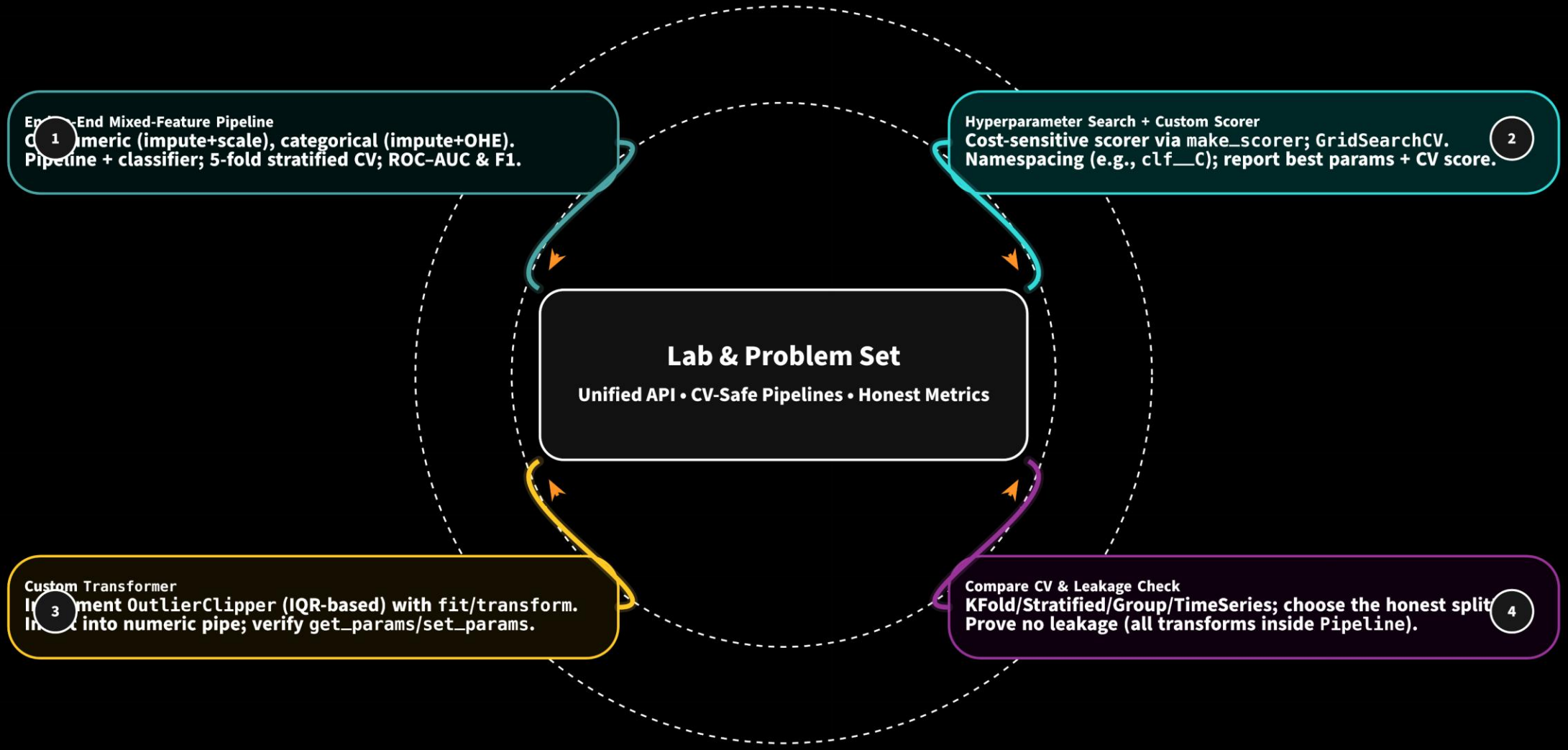
Starter.

```
from sklearn.model_selection import (KFold, StratifiedKFold, GroupKFold,
                                     TimeSeriesSplit, cross_val_score)
```

```
3
4   def eval_cv(cv, name):
5       s = cross_val_score(pipe, X, y, cv=cv, scoring="roc_auc", n_jobs=-1)
6       print(f"{name}: {s.mean():.3f}    {s.std():.3f}")
7
8   eval_cv(KFold(5, shuffle=True, random_state=0), "KFold")
9   eval_cv(StratifiedKFold(5, shuffle=True, random_state=0), "StratifiedKFold")
10  eval_cv(GroupKFold(5), "GroupKFold")           # requires groups array
11  eval_cv(TimeSeriesSplit(5), "TimeSeriesSplit") # ordered X,y
```

Leakage audit.

Prove all preprocessing is inside the pipeline and that splitters never let future or cross-group info leak into training.



Cheat Sheets & Quick Reference

Common Estimators & Methods

- **LogisticRegression** — `fit`, `predict`, `predict_proba`, `decision_function`; tune `C` (log-space), consider `class_weight='balanced'`.
- **Ridge/Lasso/ElasticNet** — regression; scale features; tune `alpha` (& `l1_ratio` for ElasticNet).
- **SVC/SVR** — kernels `linear`/`rbf`; tune `C`, `gamma`; `probability=True` for calibrated probs.
- **RandomForest**, **HistGradientBoosting** — robust to scaling; tune depth/leaves, `n_estimators`, `learning_rate` (boosting).
- **KNeighborsClassifier/Regressor** — sensitive to scale; tune `n_neighbors`, `weights`.
- **MultinomialNB/ComplementNB** — text + TF–IDF; tune `alpha`.
- **PCA/TruncatedSVD** — `fit`/`transform`; PCA dense, SVD sparse.

Metric Reference

- **ROC–AUC** (`'roc_auc'`) — threshold–free ranking; good under imbalance.
- **PR–AUC** (`'average_precision'`) — rare positives; focus on precision at recall.
- **F1 / F1-macro** (`'f1'`, `'f1_macro'`) — balance P/R; macro = class-balanced.
- **Brier score** (`'neg_brier_score'`) — probability calibration quality (more positive is better in sklearn).
- **MAE / RMSE** (`'neg_mean_absolute_error'`, `'neg_root_mean_squared_error'`) — regression; MAE robust, RMSE penalizes large errors.
- **NDCG@k** — ranking quality at top-k (use `ndcg_score`).

Param-Grid Templates

- **LogisticRegression:** `clf__C = logspace(-3,3), clf__penalty=['l2'], clf__solver=['lbfgs'].`
- **Linear SVM (text):** `clf__C = logspace(-4,2), clf__loss=['hinge','squared_hinge'].`
- **RBF SVM (tabular):** `clf__C = logspace(-2,3), clf__gamma = logspace(-4,1), clf__kernel=['rbf'].`
- **RandomForest:** `clf__n_estimators=[200,400,800], clf__max_depth=[None,6,10,14], clf__min_samples_leaf=[1,2,5].`
- **HistGradientBoosting:** `clf__learning_rate=[0.05,0.1,0.2], clf__max_depth=[None,6,10], clf__l2_regularization=[0,0.01,0.1].`
- **ColumnTransformer knobs:** `prep__num__imputer__strategy=['median','mean'], prep__cat__onehot__min_frequency=[None,5,10].`

Common Pitfalls & Anti-Patterns

- **Leakage** — fitting imputer/scaler/selector on full data. *Fix:* put *all* preprocessing inside `Pipeline`/`ColumnTransformer`.
- **Wrong CV splitter** — imbalanced/grouped/temporal data. *Fix:* `StratifiedKFold`/`GroupKFold`/`TimeSeriesSplit`.
- **Tuning on test set**. *Fix:* tune in CV (or nested CV); hold test strictly for final estimate.
- **Unscaled distance models** (KNN/SVM/SGD). *Fix:* `StandardScaler` (or `MaxAbsScaler` for sparse).
- **Dense OHE for huge categoricals**. *Fix:* keep sparse; use `min_frequency` or hashing.
- **Parallel oversubscription**. *Fix:* set outer `n_jobs`; cap BLAS threads (`OMP_NUM_THREADS`, `MKL_NUM_THREADS`).
- **No seed & no persistence**. *Fix:* set `random_state`; save pipeline via `joblib`.

Glossary of Key Classes/Functions

- **Pipeline** — chains steps; ensures `fit`/`transform`/`predict` inside CV.
- **ColumnTransformer** — per-column preprocessing for mixed features; preserves sparsity.
- **make_pipeline**/**make_column_transformer** — shorthand constructors with auto names.
- **GridSearchCV**/**RandomizedSearchCV** — hyperparameter search; use namespacing `step__param`.
- **StratifiedKFold**/**GroupKFold**/**TimeSeriesSplit** — CV for class balance, groups, and time order.
- **cross_validate**/**cross_val_score** — CV runners; pass `scoring` and `n_jobs`.
- **TransformedTargetRegressor** — wraps a regressor with an invertible target transform.

- **StandardScaler**/**MaxAbsScaler** — scaling for dense/sparse data respectively.
- **OneHotEncoder** — `handle_unknown='ignore'`, `min_frequency` for rare levels.
- **TfidfVectorizer** — text to sparse TF–IDF; tune `min_df`, `ngram_range`.
- **CalibratedClassifierCV** — probability calibration for margin models.
- **make_scorer** — wrap custom cost/metric into `scoring=`.