XGBoost

XGBoost is the gradient-boosting library behind countless winning models, covered here from training and tuning to early stopping.

Data: DMatrix vs the sklearn API

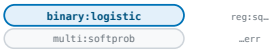
Two front doors to the same engine: the native DMatrix plus xgb.train, or XGBClassifier/XGBRegressor that take plain arrays.

PURPOSE	COMMAND	RESULT
Wrap arrays in the native container.	`dtrain = xgb.DMatrix(X_train, label=y_train)`	
Faster hist data container.	`dtrain = xgb.QuantileDMatrix(X_train, label=y_train)`	
Or skip DMatrix: the sklearn API.	`clf = xgb.XGBClassifier()` `clf.fit(X_train, y_train)`	
Add a validation set to watch.	`dvalid = xgb.DMatrix(X_valid, label=y_valid)`	
Carry feature names / weights.	`xgb.DMatrix(X, label=y, weight=w, feature_names=cols)`	

Native path: wrap arrays in xgb.DMatrix and train with xgb.train; or let XGBClassifier/XGBRegressor take plain arrays directly

Train: classifier and regressor

One engine, two estimators: XGBClassifier for labels, XGBRegressor for numbers, both .fit / .predict. The native path is xgb.train(params, dtrain, num_boost_round).

PURPOSE	COMMAND	RESULT
Train a classifier (sklearn API).	`clf = xgb.XGBClassifier(n_estimators=300, tree_method="hist").fit(X_train, y_train)`	
Train a regressor (sklearn API).	`reg = xgb.XGBRegressor(objective= "reg:squarederror").fit(X_train, y_train)`	
Train via the native API.	`bst = xgb.train(params, dtrain, num_boost_round=300)`	
Pick the objective (task).	`params = {"objective": "binary:logistic", "eval_metric": "logloss"}`	
Use CPU or GPU.	`xgb.XGBClassifier(tree_method="hist", device="cuda")`	
Same engine, two estimators.	`clf.predict(X) reg.predict(X)`	

XGBClassifier / XGBRegressor wrap the same boosting engine; xgb.train(params, dtrain, num_boost_round) returns the native Booster

Key hyperparameters

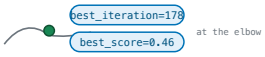
The three knobs that decide capacity: how many trees (n_estimators), how deep each tree (max_depth), and how much each tree counts (learning_rate). Lower rate plus more trees usua

PURPOSE	COMMAND	RESULT
Number of boosting rounds (trees).	`xgb.XGBClassifier(n_estimators=500)`	
Tree complexity (depth).	`xgb.XGBClassifier(max_depth=6)`	
Shrinkage per tree.	`xgb.XGBClassifier(learning_rate=0.05)`	
Subsample rows / columns.	`xgb.XGBClassifier(subsample=0.8, colsample_bytree=0.8)`	
Regularize leaf weights.	`xgb.XGBClassifier(reg_lambda=1.0, min_child_weight=1)`	
Old-style native aliases (avoid in sklearn API).	`nativeparams={"eta": 0.05, "lambda": 1.0}`	

n_estimators, max_depth, learning_rate are the three capacity knobs; use the underscored sklearn names, not the native eta / lambda / alpha aliases

Early Stopping

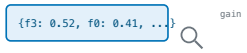
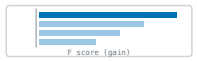
Let a validation set choose the tree count. Set early_stopping_rounds in the constructor and pass eval_set to fit; training halts when the metric stops improving, and best_iteration is recor

PURPOSE	COMMAND	RESULT
Turn on early stopping (sklearn API).	`xgb.XGBClassifier(n_estimators=1000, early_stopping_rounds=20, eval_metric="logloss")`	
Provide the watch set in fit.	`clf.fit(X_train, y_train, eval_set=[(X_valid, y_valid)])`	
Read where it stopped.	`clf.best_iterationand clf.best_score`	
Native-API early stopping.	`bst =xgb.train(params, dtrain, num_boost_round=1000, evals=[(dvalid, "valid")],early_stopping_rounds=20)`	
Callback form (save the best).	`xgb.callback.EarlyStopping(rounds=20, save_best=True)`	
Deprecated: passing it in fit (sklearn).	~~`clf.fit(..., early_stopping_rounds=20, eval_metric=...)`~~	

Set a generous n_estimators, put early_stopping_rounds + eval_metric in the constructor, pass eval_set to fit; read best_iteration / best_score

Feature Importance (gain)

Which features did the trees actually use? gain is the default and most informative; weight and cover are alternatives.

PURPOSE	COMMAND	RESULT
Normalized importances (sklearn API).	`clf.feature_importances_`	
Raw gain per feature (native scores).	`clf.get_booster()` `.get_score(importance_type="gain")`	
Compare importance types.	`get_score(importance_type="weight")` `and "cover"`	
Plot importance directly.	`xgb.plot_importance(clf,` `importance_type="gain", max_num_features=10)`	
Draw a single tree.	`xgb.plot_tree(clf, num_trees=0)`	

feature_importances_ gives normalized scores; get_score(importance_type="gain") gives raw loss reduction per split, usually the best ranking

Persist: save & load a booster

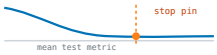
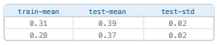
Train once, serve many times: save to portable .json or compact .ubj, reload in another process.

PURPOSE	COMMAND	RESULT
Save the fitted model (sklearn API).	`clf.save_model("model.json")`	
Save as compact binary (UBJSON).	`clf.save_model("model.ubj")`	
Load it back into an estimator.	`clf2 = xgb.XGBClassifier()` `clf2.load_model("model.json")`	
Save / load a native Booster.	`bst.save_model("booster.ubj")` `xgb.Booster().load_model("booster.ubj")`	
Deprecated: pickle / old .bin.	~~`pickle.dump`~~ `(bst, ...) / ".bin"`	
Stable, version-checked formats.	`.json` = text / `.ubj` = binary	

save_model("model.json"/".ubj") and load_model reload a model in another process; avoid pickle and the legacy .bin path

Cross-validation

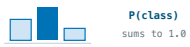
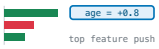
xgb.cv runs k-fold CV and, with early stopping, finds a good num_boost_round honestly.

PURPOSE	COMMAND	RESULT
K-fold CV on a DMatrix.	`cv = xgb.cv(params, dtrain, num_boost_round=500,nfold=5)`	
Stratified folds for classification.	`xgb.cv(params, dtrain, nfold=5, stratified=True)`	
CV with early stopping.	`xgb.cv(params, dtrain, nfold=5, early_stopping_rounds=20)`	
Read the results table.	`cv # a pandas DataFrame`	
Or use sklearn cross-validation.	`from sklearn.model_selection import cross_val_score` `cross_val_score(clf, X, y, cv=5)`	
Pick num_boost_round honestly.	`best_round = len(cv)`	

xgb.cv returns a tidy per-round train/test DataFrame; add stratified and early_stopping_rounds to pick num_boost_round defensibly

Predict & Explain (SHAP-style)

Get labels, probabilities, or per-feature contributions. predict_contribs returns SHAP values straight from the booster so each prediction breaks down into a base value plus one signed pu

PURPOSE	COMMAND	RESULT
Predict labels.	`y_pred = clf.predict(X_test)`	
Predict probabilities.	`proba = clf.predict_proba(X_test)`	
SHAP-style contributions (native).	`bst.predict(dtest,pred_contribs=True)`	
Use the SHAP library directly.	`import shap; shap.TreeExplainer(clf).shap_values(X_test)`	
Explain one prediction.	`bst.predict(dtest,pred_contribs=True)[i]`	
Score on a test set.	`clf.score(X_test, y_test)`	

predict gives labels, predict_proba gives probabilities; pred_contribs=True returns SHAP values: a base value plus one signed push per feature