API Guide

This guide provides a practical overview of the Tree Machine API with runnable examples for each main component.

Table of Contents

ClassifierCV

`ClassifierCV` performs automated classification with Bayesian hyperparameter optimization. It supports XGBoost and CatBoost backends.

Construction

```
from tree_machine import ClassifierCV, default_classifier
from sklearn.model_selection import StratifiedKFold

classifier = ClassifierCV(
    metric="f1_macro",           # Metric to optimize: f1, f1_macro, precision,
recall, etc.
    cv=StratifiedKFold(n_splits=5),
    n_trials=50,                 # Number of Optuna trials
    timeout=300,                 # Timeout in seconds (5 min)
    config=default_classifier,
    backend="xgboost",           # "xgboost" (default) or "catboost"
)
```

Fit and Predict

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

X, y = make_classification(n_samples=1000, n_features=20, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

classifier.fit(X_train, y_train)

# Class labels
predictions = classifier.predict(X_test)

# Probabilities (for binary: P(class=1), for multiclass: P per class)
probabilities = classifier.predict_proba(X_test)
```

SHAP Explanations

```
# Returns dict with "mean_value" and "shap_values"
explanations = classifier.explain(X_test)
print(explanations["shap_values"].shape)  # (n_samples, n_features) or
(n_samples, n_features, n_classes)
```

Attributes After Fit

```
# Best hyperparameters found by Optuna
print(classifier.best_params_)

# CV scores for the best model (per fold)
print(classifier.cv_results)

# Feature importances from the underlying model
print(classifier.feature_importances_)

# Feature names (when X is a DataFrame)
print(classifier.feature_names_)
```

RegressionCV

`RegressionCV` performs automated regression with the same optimization workflow.

Construction

```
from tree_machine import RegressionCV, default_regression
from sklearn.model_selection import KFold

regressor = RegressionCV(
    metric="mse",                # mse, mae, mape, median
    cv=KFold(n_splits=5),
    n_trials=50,
    timeout=300,
    config=default_regression,
    backend="xgboost",
)
```

Fit and Predict

```
from sklearn.datasets import make_regression

X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

regressor.fit(X_train, y_train)
predictions = regressor.predict(X_test)
```

SHAP Explanations

```
explanations = regressor.explain(X_test)
# explanations["mean_value"]: baseline
# explanations["shap_values"]: (n_samples, n_features)
```

QuantileCV

`QuantileCV` extends `RegressionCV` to predict a specific quantile (e.g., 90th percentile). It uses the pinball loss internally and shares `RegressionCVConfig`.

Construction

```
from tree_machine import QuantileCV, default_regression
from sklearn.model_selection import KFold

# Predict the 90th percentile
quantile_regressor = QuantileCV(
    alpha=0.9,                    # Quantile level (0 < alpha < 1)
    cv=KFold(n_splits=5),
    n_trials=50,
    timeout=300,
    config=default_regression,   # Same config as RegressionCV
    backend="xgboost",
)
```

Fit and Predict

```
quantile_regressor.fit(X_train, y_train)
quantile_predictions = quantile_regressor.predict(X_test)  # 90th percentile
estimates
```

Multiple Quantiles

```
# Fit separate models for median (0.5) and 90th percentile (0.9)
median_model = QuantileCV(alpha=0.5, cv=KFold(5), n_trials=30, timeout=180,
config=default_regression)
p90_model = QuantileCV(alpha=0.9, cv=KFold(5), n_trials=30, timeout=180,
config=default_regression)

median_model.fit(X_train, y_train)
p90_model.fit(X_train, y_train)

median_pred = median_model.predict(X_test)
p90_pred = p90_model.predict(X_test)
```

Configuration Objects

ClassifierCVConfig / RegressionCVConfig

Use these to customize monotonicity, interactions, and hyperparameter search space:

```
from tree_machine import ClassifierCVConfig, RegressionCVConfig, OptimizerParams

custom_config = ClassifierCVConfig(
    monotone_constraints={"age": 1, "risk_score": -1},  # 1=increasing,
-1=decreasing
    interactions=[["age", "income"], ["risk_score", "age"]],  # Allowed
interactions
    n_jobs=-1,                           # Parallel jobs (-1 = all cores)
    parameters=OptimizerParams(),        # Search space
    return_train_score=True,
)
```

Pre-configured Setups

```
from tree_machine import default_classifier, balanced_classifier,
default_regression, balanced_regression

# Default: full hyperparameter grid (performance-focused)
clf = ClassifierCV(metric="f1_macro", cv=..., n_trials=50, timeout=300,
config=default_classifier)

# Balanced: more regularization, fewer params (better generalization)
clf = ClassifierCV(metric="f1", cv=..., n_trials=50, timeout=300,
config=balanced_classifier)
```

Custom Metrics

Pass a callable with signature `(y_true, y_pred) -> float`:

```
from sklearn.metrics import accuracy_score, r2_score

# Classification
clf = ClassifierCV(
    metric=accuracy_score,
    cv=StratifiedKFold(5),
    n_trials=30,
    timeout=180,
    config=default_classifier,
)

# Regression: metrics are minimized. For R², negate: neg_r2 = lambda y_true,
y_pred: -r2_score(y_true, y_pred)
def neg_r2(y_true, y_pred):
    from sklearn.metrics import r2_score
    return -r2_score(y_true, y_pred)

reg = RegressionCV(
    metric=neg_r2,
    cv=KFold(5),
    n_trials=30,
    timeout=180,
    config=default_regression,
)
```

Custom metrics must:

- Accept `(y_true, y_pred)` and return a float.
- **Classification**: metrics are maximized (higher is better).
- **Regression**: metrics are minimized (lower is better). Negate metrics like R^2 where higher is better.

Constraints and Advanced Options

Monotonicity Constraints

Enforce monotonic relationships between features and the target:

```
custom_config = ClassifierCVConfig(
    monotone_constraints={"age": 1, "discount": -1},  # age↑ → outcome↑,
discount↑ → outcome↓
    interactions=[],
    n_jobs=-1,
    parameters=OptimizerParams(),
    return_train_score=True,
)

classifier = ClassifierCV(metric="f1_macro", cv=..., n_trials=50, timeout=300,
config=custom_config)
classifier.fit(X_train, y_train)
```

Use column names when `X` is a DataFrame; otherwise indices.

Interaction Constraints

Restrict which features can interact in splits:

```
custom_config = ClassifierCVConfig(
    monotone_constraints={},
    interactions=[["f1", "f2"], ["f3", "f4"]],  # Only these pairs can interact
    n_jobs=-1,
    parameters=OptimizerParams(),
    return_train_score=True,
)
```

Custom Optimization Parameters

```
from tree_machine import OptimizerParams

custom_params = OptimizerParams(
    n_estimators=(100, 1000),
    max_depth=(3, 10),
    learning_rate=(0.01, 0.3),
    subsample=(0.8, 1.0),
    colsample_bytree=(0.8, 1.0),
    min_child_weight=(1, 10),
    gamma=(0.0, 1.0),
    reg_alpha=(0.0, 1.0),
    reg_lambda=(0.0, 1.0),
)

config = ClassifierCVConfig(
    monotone_constraints={},
    interactions=[],
    n_jobs=-1,
    parameters=custom_params,
    return_train_score=True,
)
```

Using a Validation Set

Pass a validation set to `fit()` to optimize on it instead of CV:

```
classifier = ClassifierCV(
    metric="f1_macro",
    cv=StratifiedKFold(5),  # Ignored when validation set is provided
    n_trials=50,
    timeout=300,
    config=default_classifier,
)

classifier.fit(
    X_train, y_train,
    X_validation=X_val,
    y_validation=y_val,
)
```

Complete Workflow Example

```
import pandas as pd
from sklearn.model_selection import StratifiedKFold, train_test_split
from sklearn.metrics import classification_report
from tree_machine import ClassifierCV, default_classifier

# Load or create data
X, y = ...  # Your features and target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
stratify=y, random_state=42)

# Create and fit
clf = ClassifierCV(
    metric="f1_macro",
    cv=StratifiedKFold(n_splits=5, shuffle=True, random_state=42),
    n_trials=100,
    timeout=600,
    config=default_classifier,
    backend="xgboost",
)
clf.fit(X_train, y_train)

# Evaluate
y_pred = clf.predict(X_test)
print(classification_report(y_test, y_pred))
print(f"Best params: {clf.best_params_}")
print(f"CV scores: {clf.cv_results}")

# Explain
explanations = clf.explain(X_test)
shap_values = explanations["shap_values"]
```

Predefined Metrics

Classification

Metric	**Description**
`f1`, `f1_macro`, `f1_micro`, `f1_weighted`	F1 score variants
`precision`, `precision_macro`, `precision_micro`, `precision_weighted`	Precision variants
`recall`, `recall_macro`, `recall_micro`, `recall_weighted`	Recall variants

Regression

Metric	Description
`mse`	Mean squared error
`mae`	Mean absolute error
`mape`	Mean absolute percentage error
`median`	Median absolute error
`quantile`	Mean pinball loss (used internally by QuantileCV)