ClassifierCV

`mod` classifier_cv

Definition for ClassifierCV.

`class` ClassifierCV

Bases: `BaseAutoCV`, `ClassifierMixin`, `ExplainerMixIn`

Defines an auto classification tree, based on the bayesian optimization base class.

Source code in `src/tree_machine/classifier_cv.py`

▼ Details

```
14414514614714814915015115215315415515615715815916016116216316416516616716816917017117217317417517617717
```

`attr` **scorer** `property`

```
scorer
```

Returns correct scorer to use when scoring with RegressionCV.

`meth` **__init__**

```
__init__(metric, cv, n_trials, timeout, config)
```

Constructor for ClassifierCV.

Parameters:

Name	Type	Description	Default
`metric`	`AcceptableClassifier`	Loss metric to use as base for estimation process.	*required*
`cv`	`BaseCrossValidator`	Splitter object to use when estimating the model.	*required*
`n_trials`	`NonNegativeInt`	Number of optimization trials to use when finding a model.	*required*
`timeout`	`NonNegativeInt`	Timeout in seconds to stop the optimization.	*required*
`config`	`ClassifierCVConfig`	Configuration to use when fitting the model.	*required*

Source code in `src/tree_machine/classifier_cv.py`

```python
@validate_call(config={"arbitrary_types_allowed": True})
def __init__(
    self,
    metric: AcceptableClassifier,
    cv: BaseCrossValidator,
    n_trials: NonNegativeInt,
    timeout: NonNegativeInt,
    config: ClassifierCVConfig,
) -> None:
    """
    Constructor for ClassifierCV.

    Args:
        metric: Loss metric to use as base for estimation process.
        cv: Splitter object to use when estimating the model.
        n_trials: Number of optimization trials to use when finding a model.
        timeout: Timeout in seconds to stop the optimization.
        config: Configuration to use when fitting the model.
    """
    super().__init__(metric, cv, n_trials, timeout)
    self.config = config
```

`meth` explain

```python
explain(X, **explainer_params)
```

Explains the inputs.

Source code in `src/tree_machine/classifier_cv.py`

```python
def explain(self, X: Inputs, **explainer_params) -> dict[str,
NDArray[np.float64]]:
    """
    Explains the inputs.
    """
    check_is_fitted(self, "model_", msg="Model is not fitted.")

    if getattr(self, "explainer_", None) is None:
        self.explainer_ = TreeExplainer(self.model_, **explainer_params)

    shap_values = self.explainer_.shap_values(self._validate_X(X))
    shape = shap_values.shape

    return {
        "mean_value": self.explainer_.expected_value,
        "shap_values": shap_values.reshape(shape[0], shape[1], -1),
    }
```

`meth` fit

```python
fit(X, y, **fit_params)
```

Fits ClassifierCV.

Parameters:

Name	Type	Description	Default
`X`	`Inputs`	input data to use in fitting trees.	*required*
`y`	`GroundTruth`	actual targets for fitting.	*required*

Source code in `src/tree_machine/classifier_cv.py`

```python
def fit(self, X: Inputs, y: GroundTruth, **fit_params) -> "ClassifierCV":
    """
    Fits ClassifierCV.

    Args:
        X: input data to use in fitting trees.
        y: actual targets for fitting.
    """
    self.feature_names_ = list(X.columns) if isinstance(X, pd.DataFrame) else []
    constraints = self.config.get_kwargs(self.feature_names_)

    self.model_ = self.optimize(
        estimator_type=XGBClassifier,
        X=self._validate_X(X),
        y=self._validate_y(y),
        parameters=self.config.parameters,
        return_train_score=self.config.return_train_score,
        **constraints,
    )
    self.feature_importances_ = self.model_.feature_importances_

    return self
```

`meth` **predict**

```
predict(X)
```

Returns model predictions.

Source code in `src/tree_machine/classifier_cv.py`

```python
def predict(self, X: Inputs) -> Predictions:
    """
    Returns model predictions.
    """
    check_is_fitted(self, "model_", msg="Model is not fitted.")
    return self.model_.predict(self._validate_X(X))
```

`meth` **predict_proba**

```
predict_proba(X)
```

Returns model probability predictions.

Source code in `src/tree_machine/classifier_cv.py`

```python
def predict_proba(self, X: Inputs) -> Predictions:
    """
    Returns model probability predictions.
    """
    check_is_fitted(self, "model_", msg="Model is not fitted.")
    return self.model_.predict_proba(self._validate_X(X))
```

`class` ClassifierCVConfig

Available config to use when fitting a classification model.

dictionary containing monotonicity direction allowed for each

variable. 0 means no monotonicity, 1 means increasing and -1 means decreasing monotonicity.

interactions: list of lists containing permitted relationships in data. n_jobs: Number of jobs to use when fitting the model. parameters: dictionary with distribution bounds for each hyperparameter to search on during optimization. return_train_score: whether to return the train score when fitting the model.

Source code in `src/tree_machine/classifier_cv.py`

```
@dataclass(frozen=True, config={"arbitrary_types_allowed": True})
class ClassifierCVConfig:
    """
    Available config to use when fitting a classification model.

    monotone_constraints: dictionary containing monotonicity direction allowed
for each
        variable. 0 means no monotonicity, 1 means increasing and -1 means
decreasing
        monotonicity.
    interactions: list of lists containing permitted relationships in data.
    n_jobs: Number of jobs to use when fitting the model.
    parameters: dictionary with distribution bounds for each hyperparameter to
search
        on during optimization.
    return_train_score: whether to return the train score when fitting the
model.
    """

    monotone_constraints: dict[str, int]
    interactions: list[list[str]]
    n_jobs: int
    parameters: OptimizerParams
    return_train_score: bool

    def get_kwargs(self, feature_names: list[str]) -> dict:
        """
        Returns parsed and validated constraint configuration for a
ClassifierCV model.

        Args:
            feature_names: list of feature names. If empty, will return empty
                constraints dictionaries and lists.
        """
        return {
            "monotone_constraints": {
                feature_names.index(key): value
                for key, value in self.monotone_constraints.items()
            },
            "interaction_constraints": [
                [feature_names.index(key) for key in lt] for lt in
self.interactions
            ],
            "n_jobs": self.n_jobs,
        }
```

`meth` get_kwargs

```
get_kwargs(feature_names)
```

Returns parsed and validated constraint configuration for a ClassifierCV model.

Parameters:

Name	Type	Description	Default
`feature_names`	`list[str]`	list of feature names. If empty, will return empty constraints dictionaries and lists.	*required*

Source code in `src/tree_machine/classifier_cv.py`

```python
def get_kwargs(self, feature_names: list[str]) -> dict:
    """
    Returns parsed and validated constraint configuration for a ClassifierCV model.

    Args:
        feature_names: list of feature names. If empty, will return empty
            constraints dictionaries and lists.
    """
    return {
        "monotone_constraints": {
            feature_names.index(key): value
            for key, value in self.monotone_constraints.items()
        },
        "interaction_constraints": [
            [feature_names.index(key) for key in lt] for lt in self.interactions
        ],
        "n_jobs": self.n_jobs,
    }
```