Package TREE4

June 2024

TREE4 Class

Description This is a class container filled with methods to build decision trees using CART methodology and other novel methodologies.

Usage

```
TREE4(y, features, features_names, n_features, n_features_names,
impurity_fn, user_impur, problem, method, twoing,
min_cases_parent, min_cases_child, min_imp_gain, max_level)
```

Arguments

- *y*: the target variable as a dictionary, list, pandas DataFrame, Series or Numpy array
- *features*: numerical predictors in a dictionary, pandas DataFrame or Numpy ndarray
- *features_names*: name of numerical predictors in an iterable container like a list
- *n_features*: nominal predictors
- *n_features_names*: name of nominal predictors
- *impurity_fn*: input as a string defining the impurity function to use. For a regression problem with CART can use "between_variance". For classification with CART ["gini", "entropy"]. For FAST, TWO-STAGE or LATENT-BUDGET-TREE with regression can use "pearson" and for classification "tau". There is also the possibility for "user_defined". (default: "between_variance")
- *user_fn*: a user defined impurity function. Please note the architecture looks to find the maximum, so choose an impurity function where the best split can be found by maximisation. (default: None)
- *problem*: specify the type of problem ["regression","classification"] (default: "regression")
- *method*: specify which algorithm to use for finding the best split at each node ["CART", "FAST", "TWO-STAGE", "LATENT-BUDGET-TREE"] (default: "CART")

- *twoing*: Allows a separate CART method for handling multi-class y responses. This will find every possible two class combination of the classes, and for each find its best split according to the method. This can also be used for regression where the numerical responses are treated as ordinal variables, and the same process ensues. (default: False)
- *min_cases_parent*: the minimum number of units in each father node. (default: 10)
- *min_cases_child*: the minimum number of units in each child node. (default: 5)
- *min_imp_gain*: the minimum impurity gain after a split. (default: 0.01)
- *max_level*: define the maximum level the tree can grow to. For instance max_level = 1, will give a stump, max_level two will have maximum 4 leaves. (default: 10)

Value Returns an object of TREE4 type populated with initialised state.

Example

```
tree = TREE4(y,features,features_names,n_features,n_features_names,
impurity_fn = "between_variance", user_impur = False,
problem = "regression", method = "FAST", twoing = False,
min_cases_parent = 10, min_cases_child = 5, min_imp_gain = 0.0001,
max_level = 10)
```

TREE4.impur

Description This function calculates the impurity for each possible node/split per the method given. For a regression problem in CART it calculates the so called between_variance. For the classification problem for CART it calculates the gini index or entropy. For the novel approaches tau an pearson values can be used. Or else user_defined methods can be passed in the user_impur parameter.

- *between_variance*: $SSB = \sum_{i=1}^{G} n_i(\bar{x} - \bar{x_i})^2 \approx \bar{x_i}^2 n_i$, where n_i is the amount of samples in the group, to provide weighting to the value. With binary splitting G = 2.

- *gini*: $GW = \sum_{i=1}^{C}(\frac{n_i}{n})^2 * n_i$, where C are the amount of classes. This is not returned by definition as 1 - G, as it is a maximisation problem, so the value closest to 1, which would give a gini value of 0, which signifies perfect quality, is the one selected.
- *entropy*: $H = -\sum_{i=1}^{C} \frac{n_i}{n} log_2 \frac{n_i}{n}$
- *pearson*: $SSW = \sum_{i=1}^{n}(x_i - \bar{x})^2$
  $\eta^2_{Y|s} = 1 - \frac{SSW_{Y|s}}{SST}$ where $SST = \sum_{i=1}^{n}(x_i - \bar{x})^2$
- *tau*: $G = \sum_{i=1}^{C}(\frac{n_i}{n})^2$
  $\tau_{Y|s} = \frac{\frac{G_l n_l}{n_p} + \frac{G_r n_r}{n_p} - G_p}{1 - G_p}$

Please note, tau and pearson values have later processing to make them into their correct form, when searching for the best split, but the values that will be returned if you use the function in the library are the SSW and G respectively.

Usage

```
TREE4.impur(node)
```

Arguments

- *node*: object of MyClassNode
- *display*: boolean, used for printing the values with the non maximising value, but the true value, it is used in gini only.

Value Returns the impurity value of the given node using the given impurity function.

Example

```
for node in tree.get_all_node():
    print(node.name, tree.impur(node))
```

TREE4.__node_search_split

Description A method that searches for the best split, given the appropriate method and impurity function.

Usage

```
TREE4.__node_search_split(node, max_k, combination_split, max_c)
```

Arguments

- *node*: the current node that is looking to split
- *max_k*: Used for TWO-STAGE and LATENT-BUDGET-TREE methods to look at the max_k top ordered variables - counted only if a split is found without error (default: 1)
- *combination_split*: used for LATENT-BUDGET-TREE to indicate that the predictor classes for the respective observation be combined, and used to find a new class system. (default: False)
- *max_c*: how many subsequent predictors the current selected ordered_variable is to be combined with. Note once the bottom variable is selected to be combined with once, the process stops (default: 1)

Value Returns a tuple with three elements, the first element is the predictor name, the second is the split value, and the third is the impurity value for that split. The impurity value is mostly the addition of the impurity of the two children formed from the tree, but depending on the method and impurity_fn, for example the tau and pearson return the values for each split as given in the impur section section as η^2 and τ.

Example

```
tree._TREE4__node_search_split(node1, max_k = 1, combination_split
= False, max_c = 1)
```

TREE4.growing_tree

Description This method builds the maximum expanded tree following the chosen methodology.

Usage

```
tree.growing_tree(root,rout,proportion_total, max_k,
combination_split, max_c)
```

Arguments

- *root*: the root node
- *rout*: the path it follows (default: "start") - rout can also take values ["left", "right"]
- *proportion_total*: the maximum total proportion of explained deviance the tree has to reach (default: 0.9)
- *max_k*: Used for TWO-STAGE and LATENT-BUDGET-TREE methods to look at the max_k top ordered variables - counted only if a split is found without error (default: 1)
- *combination_split*: used for LATENT-BUDGET-TREE to indicate that the predictor classes for the respective observation be combined, and used to find a new class system. (default: False)
- *max_c*: how many subsequent predictors the current selected ordered_variable is to be combined with. Note once the bottom variable is selected to be combined with once, the process stops (default: 1)

Arguments None - just alters objects within the class.

Example

```
tree.growing_tree(mynode, "start", 0.9)
```

TREE4.identify_subtrees

Description This method associates each node with its children, grandchildren etc.

Usage

```
tree.identify_subtrees(father, leaves)
```

Arguments

- *father*: list of nodes to use as root node for growing the sub-tree
- *leaves*: list of child nodes

Value Return a dictionary having as keys the fathers and as values the children/grandchildren as NodeClass objects.

Example

```
new_dict = tree.identify_sub trees(tree.get_all_nodes(),
tree.get_leaf())
```

TREE4.alpha_calculator

Description This method returns the minimum alpha parameter for either regression or classification problem. As the tree is pruned, the dictionary becomes smaller, allowing the propagation.

For a regression problem: $\alpha = \frac{SST_p - \sum_{c=1}^{n_{children}} SST_c}{n_{children} - 1}$

For a classification problem: $\alpha = \frac{n_p[i \neq j] - \sum_{c=1}^{n_{children}} n_c[i \neq j]}{n_{children} - 1}$

where j is the majority class of that node, and i is an observation. So we are counting the amount of observations that are not in the major class at each node.

Usage

```
tree.alpha_calculator(dict)
```

Arguments

- *dict*: the dictionary returned by the TREE4.identify_subtrees() method.

Value Returns a tuple of length three, where the first element is the alpha value (cost complexity coefficient), the node where its children will be pruned, and the deviance that was reduced by allowing those children to be part of the tree.

Example

```
cut = tree.alpha_calculator(new_dict)
```

TREE4.pruning

paragraphDescription This method prunes the tree based on the cost complexity value alpha. May include other methods later.

Usage Call this function after tree.growing_tree()

```
tree.pruning(features_test, n_features_test, y_test, table, merge_leaves)
```

Arguments

- *features_test*: the numerical test features (must match the predictors that are used for training)
- *n_features_test*: nominal test features
- *y_test*: the response test values
- *png_name*: The name of the png file saved to the directory. (default: "TREE4_tree_pruned.png")
- *dot_name*: The name of the dot file saved to the directory, redundant. (default: "tree_pruned.dot")
- *table*: boolean to return a pandas DataFrame for the pruned tree
- *html*: Boolean for whether to save and show the image as a html. (default: False)
- *print_render*: Boolean for whether to display the AnyTree tree_render tree. (default: False)
- *merge_leaves*: will prune children if of exact same value, most effective in a classification tree (default: False)
- *graph_result*: Boolean for whetehr to show the leaves versus error for the pruned trees. (default: False)

- *print_tree*: Boolean, whether to print the tree with the lowest error value. (default: False)
- *visual_pruning*: Boolean, whether to print the tree with visual pruning. (default: False)

Value Returns a list of the ordered pruning values with respect to alpha, the split that is undone and the deviance added by cutting the nodes under the undone split.

Example

```
alpha, table = tree.pruning(features_test, n_features_test,
y_test, table = True)
```

TREE4.cut_tree

Description A private method that cuts the tree until there are a given amount of leaves (or close to) left on the tree.

Usage Call this function after the pruning.

```
tree.cut_tree(total_leaves)
```

Arguments

- *total_leaves*: number of leaves that have to remain.

Value Returns two lists all_node and leaves, which can be passed to tree.print_tree() to display.

Example

```
all_node, leaves = tree.cut_tree(total_leaves = 7)
#after the cut the tree has only 7 terminal nodes.

tree.print_tree(all_node = all_nodes, leaf = leaves, table =
False, html = False)
```

TREE4.print_tree

Description Print a visual representation of the formed tree showing the structure of the tree, the split, the mean (regression) or most common value (classification), the impurity and amount of samples associated with that node.

Usage

```
tree.print_tree(all_node, leaf, filename, treefile, table, html,
print_render, visual_pruning, merge_leaves)
```

Arguments

- *all_node*: A list of nodes similar to that received from tree.get_all_node() (default = None)
- *leaf*: A list of leaves similar to the received from tree.get_leaf() (default = None)
- *filename*: The of the .png file saved to your directory. Is still used when saving html = True (default: "TREE4_tree.png")
- *treefile*: Name of the pydot file saved to your directory, currently redundant (default: "tree.dot")
- *table*: Boolean response for table representation of the decision tree to be returned by the function (default: False)
- *html*: Boolean response for whether to visualise the decision tree in your browser (only checked in Chrome) (default: False)
- *print_render*: Boolean response whether to use the anytree.TreeRender to visualise the tree (default: False)
- *visual_pruning*: Whether to show the visual_pruning representation of the tree, where the branches show the amount of deviance / variance reduced by the split. (default: False)
- *merge_leaves*: will prune the above split if two terminal nodes have the exact same value. (default: False)

Value Shows a graphic representation of the tree in your window, and returns the table of the tree if selected.

Example

```
table = tree.print_tree(html = False, table = Tree)
```

TREE4.pred_x

Description Provides a prediction for the y value based on evaluating the splits for the tree for the given data set. Fits values to the model.

Usage

```
tree.pred_x(node, x, all_node, leaves)
```

Arguments

- *node*: the node of the tree, initialised at root node
- x: the new set of values, including feature names, normally as a dictionary
- *all_node*: A list of nodes similar to that received from tree.get_all_node()
- *leaves*: A list of leaves similar to tree.get_leaf()

Value Returns the predicted leaf node, which node includes the indices that can be assessed. The predicted values can be found in tree.prediction_cat for classification and tree.prediction_reg for regression. The latest prediction can look at the last values of the list i.e. tree.prediction_cat[-1].

Example

```
tree.pred_x(node1, x_to_predict, tree.get_all_node(),
tree.get_leaf())
```

TREE4.merge_leaves

Description Merges leaves for classification trees that have the same class distinction in both, i.e. undoes the spilt. Will work for regression trees but requires the values of both classes to be identical.

Usage Call this function after the growing_tree, but generally used with print_tree.

```
merge_leaves(all_node, leaves )
```

Arguments

- *all_node*: A list of all nodes for the tree
- *leaves*: A list of all leaves for the tree

Value Returns two lists all_node and leaves, which can be passed to tree.print_tree() to display.

Example

```
all_node, leaf = self.merge_leaves(all_node, leaf)
```

NodeClass

Description This class is for the creation of the tree nodes using the node class from the anytree library.

Usage

```
NodeClass(name, indexes, split, parent, node_level, to_pop)
```

Arguments

- *name*: the name of the node
- *indexes*: the indexes of data frame
- *split*: The split that occurred at this particular node as a string. This parameter is unused and is created by the bin_split() method (default: None)
- *parent*: The parental node to this node, generally created with the bin_split method (default: None)
- *node_level*: The level of the node in the tree (default: 0)
- *to_pop*: Boolean a flag to indicate whether to pop this node from the tree, essentially a marker that can be checked when creating node lists (default: False)

Value Returns an object of type NodeClass.

Example

```
my_tree = NodeClass('n1', indexes)
```

NodeClass.bin_split

Description bin_split is a method of NodeClass which defines the binary splitting mechanism for the the observations under investigation.

Usage

```
node.bin_split(feat, feat_nominal, var_name, threshold)
```

Arguments

- *feat*: the numerical variables
- *feat_nominal*: the nominal variables
- *var_name* : the variable in which we can find the split
- *threshold*: the splitting value/class

Value Returns the left and right node as NodeClass objects.

Example

```
left_node, right_node = node.bin_split(feat, feat_nominal, var_name,
threshold)
```

Note there is an issue with the bin_split function if it is a classification problem, and the class label has a length of 1 i.e. "A". Generally I append the name of the response to make it longer.