COBWEB

Python Implementation: Little Manual

Xin Lian
Teachable AI Lab, Georgia Tech
xinlian@gatech.edu

Nov 2022

Original Code: https://github.com/cmaclell/concept_formation

`CobwebNode(otherNode=None)`

https://concept-formation.readthedocs.io/en/latest/concept_formation.html#cobwebnode

Self Attributes:

- ***`concept_id`***: *`int`.*
  The unique name for every concept.
  Generated by `self.gensym()`.
- ***`count`***: *`float`.*
  The number of instances included in the node.
  Initialized with `0.0`.
  If `otherNode` is not None, it increments `otherNode.count` (so it is indeed initialized with `otherNode.count`).
- ***`av_counts`***: *`dict`.*
  Its keys are the attributes, and the corresponding values are also the dicts whose keys are val (values of attributes) and values are the counts of the values.
  It is indeed the attribute-value pair table for the node.
  Initialized with empty `dict {}`.
  If `otherNode` is not None, it is updated with `otherNode.av_counts` (so it is indeed initialized with `otherNode.av_counts`).
- ***`children`***: *`list`.*
  The entries of `children` are of type `CobwebNode`.
  Initialized with empty list.
  If `otherNode` is not `None`, it appends all the `child` (entries) of `children` from `otherNode`.
- ***`parent`***: *`CobwebNode`.*
  Initialized with `None`.
  If `otherNode` is not `None`, it copies the `parent` from `otherNode`.
- ***`tree`***: *`CobwebTree`.*
  Initialized with `None`.
  If `otherNode` is not `None`, it copies the `tree` from `otherNode`.

Pre-defined Attributes:

- ***`_counter`***: *int.*
  Initialized with 0, and then will be incremented by 1 after every concept is created.

`.shallow_copy():`

Create a shallow copy of the current node.
Copy the information relevant to the node's probability table (probabilistic concept) without maintaining reference to other elements of the tree, except for the root which is necessary to calculate category utility.
So it is actually generating a "preview" version of the current `CobwebNode` – it initiates another new `CobwebNode`, and copies the following from the current `CobwebNode`.

- `tree` in `self`
- `parent` in `self`
- `count` in `self`
- `av_counts` in `self`

Output: `CobwebNode`.
One of the primary functions.

`.attrs(attr_filter=None):`

Returns the "filtered" `self.av_counts`.
It returns an iterable whose entries are indeed the attributes (`str`).
iterates over the attributes present in the node's attribute-value table.

- If the `attr_filter` is `None`, then by default, the function will return the attributes that are not "hidden", i.e., the attributes that don't have "_" as their prefixes.

Returns a filter object whose entries are the (names of the) attributes (only) that can be iterated. So the iterables have type `str`.

- If `attr_filter == 'all'`, then the function will return all the attributes, i.e., the complete `self.av_counts`.
  Returns a dictionary. It doesn't matter the type of returns is not consistent. Because when dictionary is used as an iterable, only its keys are iterated. In this case, only the keys, which are the attributes, are iterated.
- Otherwise, the `attr_filter` will be recognized as a function which returns Booleans wrt entries in a sequence (typically can be a lambda function) and be applied to the predefined function `filter` in Python.
  Returns a filter object whose entries are the (names of the) attributes (only) that can be iterated, so the iterables have type `str`.

Input: None or a filter function.
Output: a filter object or a dictionary.
One of the primary functions.

`.increment_counts(instance):`

Now an `instance` (a `dict` whose keys are the attributes it has, and values are the values for the attributes) is passed to the current node. Update the counts according to the data of the instance.
`count` is incremented by 1.
And `av_counts` is updated just like what `self.update_counts_from_node` does.
Input: `dict`.
Output: nothing.
One of the primary functions.

`.update_counts_from_node(node):`

Increments the counts of attributes from another `CobwebNode`, `node` to the current node (`self`).

- Increment `self.count` by `node.count`
- Increment all the counts of values of the attributes in `node` to the current node. Iterate all the attributes in `node` then increment its counts correspondingly into the current one.
  Suppose in one iteration, we add the count data `attr: {val: count}` from `node` into `self`.
  - If `attr` is even not recorded in `self`, set up a new empty dictionary for it.
  - If `attr` is recorded in `self` before, and `val` is not recorded for `attr` in `self`, add the key-value pair `(val, 0)` to `attr`.
  - Otherwise, just increment `count` at the corresponding `val`.

It will be used when copying nodes or merging nodes.
Input: `CobwebNode`.
Output: nothing.
One of the primary functions.

`.expected_correct_guesses():`

Calculate the expected correct guesses within the node.
Given the formula of Category utility:

$$\frac{\sum_{k=1}^{n} P(C_k)[\sum_i \sum_j P(A_i = V_{ij}|C_k)^2 - \sum_i \sum_j P(A_i = V_{ij})^2]}{n}$$

The function is actually calculating the part:

$$\frac{\sum_{k=1}^{n} P(C_k)[\sum_i \sum_j P(A_i = V_{ij}|C_k)^2 - \boxed{\sum_i \sum_j P(A_i = V_{ij})^2}]}{n}$$

Input: None.
Output: `float`.
One of the primary functions.

`.category_utility():`

Calculate Category Utility:

$$\frac{\sum_{k=1}^{n} P(C_k)[\sum_i \sum_j P(A_i = V_{ij}|C_k)^2 - \sum_i \sum_j P(A_i = V_{ij})^2]}{n}$$

Here actually calculates the CU statistic of the subtree (where the current node is the root). `n` is the number of children of the current node. So C_k's are all the children of the current node.
Input: None.
Output: `float`.
Used for functions `cu_for_insert`, `cu_for_new_child`, `cu_for_merge`, `cu_for_fringe_split`, `cu_for_split`.

`.get_best_operation(instance, best1, best2, best1_cu, possible_ops=["best", "new", "merge", "split"]):`

An instance is given, choose the best operation having the greatest CU statistic. In the case of ties, an operator is randomly chosen.
`best1`, `best2`, and `best1_cu` are returned by `self.two_best_children(instance)`.
Input: `dict, (float, CobwebNode), (float, CobwebNode), float, list`
Returns `(float, str)`.
One of the output functions. Used to return the best operation in `CobwebTree.fit`.

`.two_best_children(instance):`

Calculates the category utility of inserting the instance into each of this node's children and returns the best two. In the event of ties children are sorted first by category utility, then by their size, then by a random value.
First calculate **`relative_cu_for_insert`** for each child, then find the 2 best child wrt **`relative_cu_for_insert`**.
Finally return the corresponding overall CU calculating by the formula (calculate CU in this formula):

$$CU_{rel}(CU) = (CU - Const) \cdot n \cdot (|C| + 1)$$

Input: `dict`.
Returns `(float, (float, CobwebNode), (float, CobwebNode))`
One of the output functions. The outputs of the function is in `get_best_operation` in `CobwebTree.cobweb` and `CobwebTree._cobweb_categorize`.

`.compute_relative_CU_const(instance):`

Compute the relative CU in the case that the instance is directly absorbed by the root.

$$\frac{\sum_{k=1}^{n} \frac{|C_k|}{|C|+1} \sum_i \sum_j P(A_i = V_{ij}|C_k)^2 - \sum_i \sum_j P(A_i = V_{ij}|C_{updated})^2}{n}$$

C: root (self)
|C|: C.count (self.count)
|C_k|: c.count
Input: `dict`.
Output: `float`.
Used in `two_best_children`.

`.relative_cu_for_insert(child, instance):`

(Used only in the function `two_best_children` to pick the "best" two cases for inserting the instance to one of the children, with a modified formula. It does NOT compute the CU for insertion operation for the subtree, see `cu_for_insert`.)
The relative CU score is more efficient to calculate for each insert operation and is guaranteed to have the same rank ordering as the CU score so it can be used to determine which insert operation is best.
The relative CU score is given by

$$CU_{rel}(CU) = (CU - Const) \cdot n \cdot (|C| + 1)$$

Where CU is given by

$$\frac{\sum_{k=1}^{n} P(C_k)[\sum_i \sum_j P(A_i = V_{ij}|C_k)^2 - \sum_i \sum_j P(A_i = V_{ij})^2]}{n}$$

And Const is given by

$$\frac{\sum_{k=1}^{n} \frac{|C_k|}{|C|+1} \sum_i \sum_j P(A_i = V_{ij}|C_k)^2 - \sum_i \sum_j P(A_i = V_{ij}|C_{updated})^2}{n}$$

Compute the relative CU in the case that the instance is inserted to the node *child*, which is a child of the current node:

$$CU_{rel,ins}(C_i) = (|C_i|+1) \cdot \sum_i \sum_j P(A_i = V_{ij}|C_{i,updated})^2 - |C_i| \cdot \sum_i \sum_j P(A_i = V_{ij}|C_i)^2$$

Input: *CobwebNode*.

Output: *float*.

Used in *two_best_children*.

.cu_for_insert(child, instance):

Compute the CU (of the subtree. Use *category_utility()*) in the case – inserting *instance* to a specified *child*.

Note that, this function does not do the job of inserting instance to child – it is just an assumption made for calculation. But I didn't find where the function is used in *CobwebNode* class :-)

Input: *CobwebNode*, *dict*

Output: *float*.

One of the output functions. I guess it may be used in *get_best_operation*.

.create_new_child(instance):

Create a new *CobwebNode* as a new child to the current node with the counts initialized by the given *instance*. The new child is initialized with:

- *parent*: *self*
- *tree*: *self.tree*
- Initialize *count* and *av_counts* according to instance only.

Further, *self.children* adds the child just generated.

Input: *dict*

Output: *CobwebNode*

Used in *cu_for_new_child* and *cu_for_fringe_split*.

.create_child_with_current_counts():

Create a new *CobwebNode* as a new child to the current node with the counts initialized by the current node's counts when there is at least one instance included in *self*. It is opposite to *create_new_child*.

This is used in the special case of a fringe split when a new mode is created at a leaf.

The new child is initialized with the whole *self*, except:

- *concept_id*: always a new one
- *parent*: *self*
- *tree*: *self.tree*
- Initialize *count* and *av_counts* according to instance only.

Further, *self.children* adds the child just generated.

Input: None

Output: *CobwebNode*

Used in *cu_for_fringe_split*.

.cu_for_new_child(instance):

Compute the CU (of the subtree. Use *category_utility()*) in the case – inserting *instance* to a newly-created *child*. Note that, this function does not do the job of inserting instance to a new child – it is just an assumption made for calculation.

Input: *dict*

Output: *float*.

Used in *get_best_operation*.

.merge(best1, best2):

Merge the two "best" nodes (evaluated by *two_best_children)*. First generates a new parent node *new_child* for these two. It is initiated with

- *new_child.parent = self* and
- *new_child.tree = self.tree*,

then remove them from *self.children* and add them to *new_child.children*.

new_child is returned.

Input: *CobwebNode*, *CobwebNode*

Output: *CobwebNode*

One of the output functions.

.cu_for_merge(best1, best2, instance):

Compute the CU (of the subtree. Use *category_utility()*) in the case – inserting *instance* to the newly-created *new_child* whose data is merged from some other "best" two children of *self*.

Note that, this function does not do the job of inserting instance to a new child – it is just an assumption made for calculation.

Input: *CobwebNode*, *CobwebNode, dict*

Output: *float*.

Used in *get_best_operation*.

.split(best):

Split the "best" node and promote its children. *best* is removed from *self.children* then all child in *best.children* is added to *self.children*.

Input: *CobwebNode*

Output: None

One of the output functions.

.cu_for_fringe_split(instance):

Compute the CU (of the subtree. Use *category_utility()*) in the case – inserting *instance* to the current node while creates two children for the current node:

- one child is initialized with *create_child_with_current_counts*,
- the other child is initialized with *create_new_child(instance).*

And the current node should be a leaf.

Note that, this function does not do the actual job– it is just an assumption made for calculation.

A fringe split is essentially a new operation performed at a leaf. It is necessary to have the distinction because unlike a normal split, a normal fringe split must also push the parent down to maintain a proper tree structure. This is useful for identifying unnecessary fringe split, when the two leaves are essentially identical. It can be used to keep the tree from growing and to increase the tree's predictive accuracy.

Input: *dict*

Output: *float*.

One of the output functions.

.cu_for_split(best):

Compute the CU (of the subtree. Use *category_utility()*) in the case – remove *best* from the children of the current node

and promote all the children of *best*'s to the children of the current one's.
Note that, this function does not do the actual job– it is just an assumption made for calculation.
Also note that, NO instance is inserted to somewhere else in this calculating process. The instance is still (pre-inserted) at the root, and after the promotions we may choose a 'best' child of the current node again if we find this is the best operation.
Input: *CobwebNode*
Output: *float*.
Used in *get_best_operation*.

.is_exact_match(instance):
Find if the current concept (node) exactly matches a given *instance*. "Match" means:

- the attributes the instance has the same with the current node's.
- the attribute values of the instance should all have been recorded in the current node,
- all the instances included in the current node should have the same values for the attributes (so the counts for every attribute in the current node is just the same as the number of instances included in the current node).

Hidden attributes are not considered.
Input: *dict*
Output: *bool*
One of the output functions.

.gensym():
Increments *_counter* by 1 in the CobwebNode class so that every generated CobwebNode has a unique *_counter*. Then every node can have its unique id.
Input: None
Output: *int*.
Used in *__init__*.

.pretty_print(depth=0):
Print the categorization tree level by level start from level *depth*.
Input: None or integer.
Output: *str*
Used in *__str__*.

.depth():
Calculate the depth of the current node in the tree.
Input: None
Output: *int*.
One of the output functions.

.is_parent(other_concept):
Find if the current concept (node) is a parent of *other_concept*. It does not include the direct parent only – it finds if the current node is a predecessor of *other_concept*.
Input: *CobwebNode*
Output: *bool*
One of the output functions.

.num_concepts():
Count the number of concepts contained below the current node (i.e. children).
Input: None
Output: *int*

One of the output functions.

.output_json():

Outputs the categorization tree in JSON form.
Input: None
Output: *dict*
One of the output functions.

.get_weighted_values(attr, allow_none=True):

Construct a probability table for the current node by calculating the probability (or weight) of each value in the given attribute *attr*.
If *allow_none = True*, the list of weights will include one more tuple for the case *None*, and the corresponding probability is defined by *(self.count - val_count) / self.count* where *val_count* is the number of all counts for the *attr*.
Input: *str*, *bool* or *None*
Output: list (with entries tuples)
Used in *predict*.

.predict(attr, choice_fn="most likely", allow_none=True):

Given the list *choices* return by *get_weighted_values(attr, allow_none)*, predict the value of the attribute *attr* by *most_likely_choice(choices)* or *weighted_choice(choices)* from *concept_information.utils*.
Input: *str*, *str* or *None*, *bool* or *None*
Output: *str*.
One of the output functions. Used in *CobwebTree.inder_missing*.

.probability(attr, val):

Returns the probability of a particular attribute value at the current concept. This considers the possibilities that an attribute can take any of the values available at the root or be missing.
If you want to check if the probability that an attribute is missing, then check for the probability that the val is ``None``.
Input: *str*, *str*
Output: *float*.
One of the output functions.

.log_likelihood(child_leaf):

Returns the log-likelihood of a leaf contained within the current concept. Note, if the leaf contains multiple instances, then it is treated as if it contained just a single instance (this function is just called multiple times for each instance in the leaf).
Input: *CobwebNode*
Output: *float*

CobwebTree()

https://concept-formation.readthedocs.io/en/latest/concept_formation.html#cobwebtree

Self Attributes:

- ***root***: *`CobwebNode`.*
  The root of the tree.
  Initialized with an empty node. Also set the tree of the root as self: `self.root.tree.self`.

.clear():

Clears the concepts (nodes) of the tree.
Input: *`None`*
Output: *`None`*
One of the output functions.

.sanity_check_instance(instance):

Sanity check for given instance.
Input: *`None`*
Output: *`None`*
Used in *`ifit`*, *`infer_missing`*, *`categorize`*

.ifit(instance):

Incrementally fit a new instance into the tree and return its resulting concept.
Input: *`dict`*
Output: *`CobwebNode`*.
Used in *`fit`*.

.fit(instances, iterations=1, randomize_first=True):

Fit a collection of instances into the tree.
This is indeed a batch version of *`ifit`* function that takes a collection of instances and categorizes all of them:

- The instances can be incorporated multiple times wit given iterations to burn in the tree with prior knowledge.
- After every iteration, the sequence of instance will be randomized (with shuffle). If *`randomize_first=True`*, the sequence will be randomized before the first iteration.

Input: *`list`* (of *`dict`*), *`int`* or None, *`bool`* or None.
Output: None.
One of the output functions.

.cobweb(instance):

The core cobweb algorithm in fitting and categorization.
At each node,

- The algorithm first calculates the category utility of inserting the instance at each of the node's children, keeping the best two (with *`CobwebNode.two_best_children`*)
- Then calculates the CU of performing other operations using the best two children (with *`CobwebNode.get_best_operation`*), committing to whichever operation results in the highest CU. In the case of ties, an operation is chosen at random.
- In the base case (i.e. a leaf node), the algorithm checks to see if the current leaf is an exact match to the current node. If it is, the instance is inserted, and the leaf is returned. Otherwise, a new leaf is created.

The final leaf with *`instance`* inserted is returned.
Input: *`dict`*
Output: *`CobwebNode`*
Used in *`ifit`*.

.cobweb_categorize(instance):

A cobweb specific version of categorize. This is not intended to be externally called.

Input: dict

Output: None

Used in *`categorize`*.

.infer_missing(instance, choice_fn="most likely", allow_none=True):

Fill the attribute values that are empty for *`instance`* with *`CobwebNode.predict`* (choice function for *`predict`* is "most likely" or "sampled") for each missing attribute.

Input: *`dict`*, *`str`* or None, *`bool`* or None

Output: *`dict`*

One of the output functions.

.categorize(instance):

Sort an instance in the categorization tree and return its resulting concept.

This is different from *`fit`* – *`fit`* is to insert one instance into the tree, while *`categorize`* does not, it just tries to classify the instance to some concept without actually inserting it.

```
self._sanity_check_instance(instance)
return self._cobweb_categorize(instance)
```

Input: *`dict`*

Output: None

One of the output functions.

cluster()

https://concept-formation.readthedocs.io/en/latest/concept_formation.html#cobwebnode

---

.cluster(tree, instances, minsplit=1, maxsplit=1, mode=True):

Categorize a list of instances into a tree and return a list of flat cluster labeling based on successive splits of the tree.

The inputs:

- *tree*: *CobwebTree*. The categorization tree used.
- *instances*: *list*. The entries are instances, which are dict.
- *minsplit*: *int* or None. The minimum number of splits to perform on the tree. You can think of it as the minimum level number of the tree that first appears leaves.
- *maxsplit*: *int* or None. The maximum number of splits to perform on the tree. You can think of it as the deepest level number of the tree, if cost-saving.
- *mode*: *bool* or None.

The output is indeed the generator returned by *cluster_iter*, except only the first element of every entry (i.e. *clusters*) is retained, and *label=True*, so the entries in clusters should be *str* only.

Output: a generator whose entries are lists of *str*.

.k_cluster(tree, instances, k=3, mod=True):

Categorize a list of instances into a tree and return a flat cluster where *len(set(clustering)) <= k*.

The function is indeed implementing *cluster_iter* and returns a list of clusters *clusters*, except that only the case that

- the tree has no more than *k* clusters, and
- the list returned in the last split (at loop *maxsplit*)

is returned.

Input: *CobwebTree*, *list* (of dict), *int* or None, *bool* or None)

Output: list (of *str* or *CobwebNode*)

.depth_labels(tree, instances, mod=True)

Categorize instances and return a list in shape *(max(conceptDepth) x len(instances))* whose entries are the labels of nodes. Entry (*i, j)* shows the node that instance *j* inserted/categorized to at level *i*.

.CU(cluster, leaves):

Calculates the Category Utility of a tree state given clusters and leaves.

The function calculates CU with *CobwebNode.category_utility(root)* after gathering all the data from the associated descendants to the associated children of the root.

Input: *list* (of *CobwebNode*)

Output: *float*

.AICc(clusters, leaves):

Calculates the Akaike Information Criterion of a given clustering from a given tree and set of instances with a correction for finite sample sizes.

Input: *list* (of *CobwebNode*)

Output: *float*

.AIC(clusters, leaves):

Calculates the Akaike Information Criterion of a given clustering from a given tree and set of instances.

Input: *list* (of *CobwebNode*)

Output: *float*

.BIC(clusters, leaves):

Calculates the Bayesian Information Criterion of a given clustering from a given tree and set of instances.

Input: *list* (of *CobwebNode*)

Output: *float*

.cluster_split_search(tree, instances, heuristic=CU, minsplit=1, maxsplit=1, mod=True, labels=True, verbose=True):

Find a clustering of the instances given the tree that is based on successive splittings of the tree to minimize some heuristic function.

When *verbose=True*, the function will print split, concept id, and heuristics for the results.

Input: *CobwebTree*, *list* (of *dict*), *function* or *str*, *int* or *None*, *int* or *None*, *bool* or *None*, *bool* or *None*, *bool* or *None*

Output: list (of *str*)

.cluster_iter(tree, instances, heuristic=CU, minsplit=1, maxsplit=100000, mod=True, labels=True):

The core clustering process that splits the tree according to a given heuristic.

The inputs of the function are:

- *tree*: *CobwebTree*. The categorization tree used.
- *instances*: *list*. The entries are instances, which are *dict*.
- *heuristic*: *function* or *str*. The evaluation metric. Can choose from functions *CU*, *AIC*, *BIC* and *AICc* or str *'CU'*, *'AIC'*, *'BIC'*, and *'AICC'*.
- *minsplit*: *int*. The minimum number of splits to perform on the tree. One "split" here means to split some node which is a child of the root.
- *maxsplit*: *int*. The maximum number of splits to perform on the tree.

The function first checks if the inputs of the function are all valid, then ascertain the heuristic type.

mod is short for modify. If *mod=True*, every instance in instances will be inserted in the tree with *CobwebTree.ifit*. Otherwise, the instance will be categorized only with *CobwebTree.categorize*. The concepts (*CobwebNode*) that the instances are inserted/categorized to are stored in the list *temp_clusters*.

Then from *minsplit* to *maxsplit*, in every loop, the function has the following:

- *clusters*: the list of all the concepts (nodes, *CobwebNode* or *str*) of the tree except its root that are associated with the instances being inserted/categorized.
  If *label=True*, the entries will be *str*: *"Concept" = str(concept_id)*. Otherwise, they are still *CobwebNode*.
  I think the duplicates are permitted in the list, so as the other lists in every loop.
- *cluster_assign*: the list of concepts being the children of the root that are associated with the instances being inserted/categorized (the concepts may include the root, if the instance is inserted/categorized to the root directly..)
- *child_cluster_assign*: the list of nodes each of which is a child of each of the entry in *cluster_assign* that is traced from.

In every loop, the function will *yield clusters* and *heuristic(cluster_assign, temp_clusters)*. So the function will finally return a generator whose entries are *clusters, heuristic(cluster_assign, temp_clusters).*

- a list of tuples *split_cus*, whose entries are like *(heuristic(c_labels, temp_clusters), i, target)* where
  - *i, target* in *enumerate(set(cluster_assign))*. So it is some node just below the root that is associated with some instance.
  - *c_labels*: a list of nodes (*CobwebNode*) which are indeed the same as *cluster_assign*, except that target is not included, and the traced child of target is included instead.

  split_cus is then sorted (wrt the first entry in ascending order, which is the value of heuristic).
  Finally, the node at the very first of *split_cus* is split with *CobwebNode.split* as it is the least cohesive cluster.

The outputs of the function: a generator whose entries are *(clusters, heuristic(cluster_assign, temp_clusters))*

So a generator with entries *(list* (of *CobwebNode* or *str*)*, float)*

Used in *cluster*, *k_cluster*, *cluster_split_search*.