
Peewit 0.10

Experiment Model and Services

Document Version 0.10.2



Contents

1	First Glance	2
1.1	Examples	2
2	Experiment Model	6
2.1	Experiment Tree	6
2.1.1	Overview	6
2.1.2	Descent Production	6
2.1.3	Ascent Production	7
2.1.4	Branching	7
2.1.5	Partial Orders and E-Trees	8
2.1.6	Production Functions	8
2.2	V-Cubes	8
2.2.1	V-Cube Dimensions	8
2.2.2	Recursive Type	8
2.2.3	Dimension Sizes and Values Labels	9
2.2.4	Side Cubes	9
2.2.5	Consistency Constraints	10
2.3	Advanced Tree Handling	11
2.3.1	Sibling Values	11
2.3.2	Pruning	11
2.3.3	Looping	11
2.4	Input Resolution	12
2.4.1	Node Relations	12
2.4.2	Input from Descent and Ascent Cubes	12
2.4.3	Resolving Input Conflicts	13
2.4.4	Setting Cube Dimensions	13
3	Services	16
3.1	V-Cube Services	16
3.1.1	V-Cube Manipulation	16
3.1.2	V-Cube Alignment	17
3.1.3	Descent Values from V-Cubes	17
3.1.4	Precomputed V-Cubes	17
3.2	Housekeeping Services	18
3.2.1	Persistent Name Space	18
3.2.2	Regime on Paths	18
3.2.3	Archiving	18
3.2.4	Parallelization	19
3.2.5	Timeouts	19
3.3	Future Services	19
3.3.1	Voluntary Type Checking	19
3.3.2	Progress Indication	20

1 First Glance

1.1 Examples

```
class rabbit(ENode):
    def descent(self):
        return [ 'a', 'b' , 'c', 'd' ]

class carrot(ENode):
    def descent(self):
        A = Tube(line=1)
        B = Tube(line=2)
        C = Tube(line=3)
        return [ A, B, C ]

class kasha(ENode):
    def descent(self, rabbit, carrot ):

        def h(x,y,z):
            return abs(("abcd".index(x)**2 - 5*z ) * y.line )

        d2 = h(rabbit, carrot, 2)
        d3 = h(rabbit, carrot, 3)

        return [ d2, d3 ]

    def explicit_descent_labels(self):
        return ["He", "Hu"]
```

Figure 1.1: Toy examples 1 taken from peewit test code, definitions of e-nodes.

```
etree = [ rabbit(),
          carrot(),
          kasha(),
          ]

E = Experiment( pathkeeper = PK, version = 0)
E.define(etree)
vc = E.run()
vc.show()
```

Figure 1.2: Toy examples 1, run: set a raw e-tree, define experiment using that tree, run it, and show results from final v-cube. The content of that v-cube is shown in figure 1.3

```

v-cube: basics--0--kasha--plain
dims:   ['rabbit', 'carrot', 'kasha']
target: kasha

rabbit: ['a', 'b', 'c', 'd']
carrot: ['A', 'B', 'C']
kasha:  ['He', 'Hu']

left dims: ['rabbit', 'carrot', 'kasha']

[[[ 10  15 ]
   [ 20  30 ]
   [ 30  45 ]]]

[[[ 9  14 ]
   [ 18 28 ]
   [ 27 42 ]]]

[[[ 6  11 ]
   [ 12 22 ]
   [ 18 33 ]]]

[[[ 1  6 ]
   [ 2 12 ]
   [ 3 18 ]]]]

```

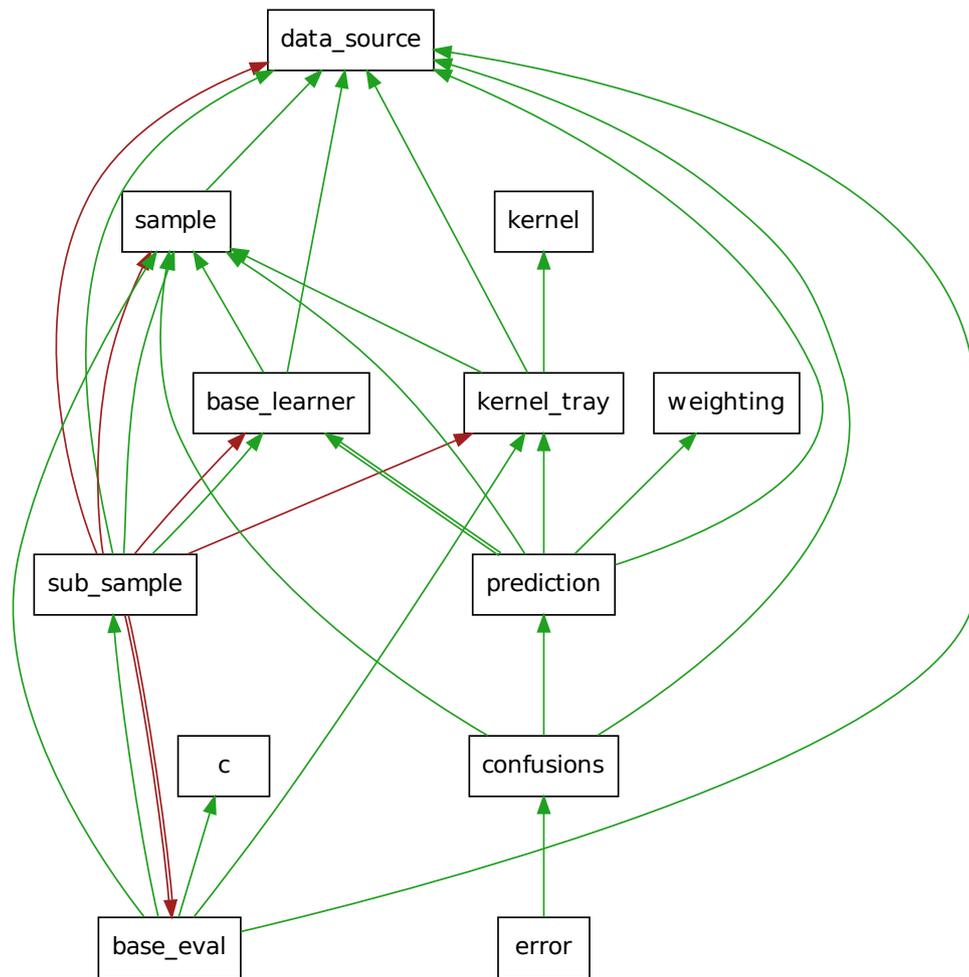
Figure 1.3: Toy examples 1, result: text-output of returned v-cube. It stems from e-node `kasha` and therefore has `kasha` as target dimension. The values produce by the e-nodes `carrot` and `rabbit` cannot be seen here as they are stored in separate cubes which are, however, link in the `kasha` cube.

```

etree = [ data_source(ts=[]),
          sample(ts=[]),
          kernel(),
          kernel_tray(),
          [ [ base_learner(),
              sub_sample(ts=[0]),
              c(ts=[0]),
              leaf(),
            ],
            [ weighting(),
              prediction(),
              confusions(),
              error()
            ]
          ]
        ]

```

Figure 1.4: Example 2, raw e-tree. This tree is taken from a multi-class experiment that probes different base learner aggregations. The raw tree does not define the dependencies between nodes but has to be consistent with respect to the dependencies. If node `b` depends on node `a` for descent `b` must come after `a` in depth-left-first order.



pwmc_26_test

Figure 1.5: Example 2, input graph. The direct dependencies/inputs of the nodes used in figure 1.4. Green lines indicate descent inputs, red lines ascent inputs, and double lines inputs from other branches or successors. Though this is more telling than the raw e-tree, we cannot see what the node do and how they interact. This is, however, much the perspective of peewit as it does not care what is happening inside nodes.

```

class sub_sample(ENode):

    def descent(self, sample, data_source, base_learner):

        data_source.positives = [ base_learner.output_j ]

        sampler = SubSampling(sequence      = data_source.get_X(),
                              output_sequence = data_source.get_YR(),
                              restriction     = sample[0],
                              output_restriction = [ base_learner.output_i,
                                                      base_learner.output_j ],
                              seed          = 1,
                              n_folds      = self.n_folds,
                              )
        return sampler.get_cv_indices()

    def ascent(self, sample, data_source, kernel_tray, base_eval, leaf):

        vcs_opt = base_eval.aggs([('argmin', 'c'),
                                ('mean', 'sub_sample'),
                                ('sel', 'leaf', 'error')])

        base_learner.c_opt = vcs_opt.get_value()
        ...

```

Figure 1.6: Example 2, inside a node. The descent function uses some class for sub-sampling. In the ascent function a v-cube passed as argument `base_learner` is aggregated for selecting an optimal parameter. The sub-sampling class is not part of peewit, the v-cube aggregation is.

2 Experiment Model

This section is dedicated to the experiment model. It defines notions *e-tree*, *e-node*, *v-cubes* and aims completeness and correctness at the expense of examples and motivation.

2.1 Experiment Tree

2.1.1 Overview

An peewit-experiment is given an rooted, ordered tree of *e-nodes*. When the experiment runs the tree is traversed in depth-left-first order where each node is visited twice, before (*descent*) and after (*ascent*) deploying the subtree below it. On descent, the *descent function* of the node is called that produces a sequences of alternative values for that node. The subtree below is deployed for each of these values in turn. This means that, if we descent a linear subtree, we obtain a value tree that has on a given level as many branches as the corresponding node yields *descent values* (though in a typical experiment many nodes give a single descent values only). We look at this value-tree a sequence of value-cubes with increasing number of dimensions. Therefore, we attach to each e-node two so-called *v-cubes* one for the descent values and another for the *ascent values*. The latter are the outputs of the *ascent function* that is called called on ascent, though the ascent production is optional and in typical experiments only defined by some of the e-nodes.

The value produced by a node depend on the values produced of other nodes, such that we speak of the (descent- and ascent-)*dependencies* of the node. The dependencies given by the *input nodes* the production functions refer to plus the indirect dependencies. For descent, any node that comes before in the *linear order* on the nodes induced by the traversal is valid input. For ascent we can also query the nodes from the subtree below. If an input node is a direct *tree ancestor* it provides a single descent value as input. It is lies on another branch or in the subtree below, the input value are all values is has produced for the given values of tree ancestors it shares with the referring node.

The final result of the experiment run is the v-cubes attached to the left-depth-most node. As it holds references to the cubes of the dependencies more or less all produced values are included. This result is a pure data-cube that does not hold any procedural information from the run.

2.1.2 Descent Production

Each e-node has a *descent function* that produces batch sequence of *descent values* of arbitrary type. The function comes with a set of *descent input names* that refer to other e-nodes, the *input nodes* (if there are branches, two or more e-nodes may have identical names, we explain in 2.4.1 how the references are resolved in this case). The *dependencies* of an e-nodes are the *input nodes* plus their dependencies in turn. In subsection 2.4.4 we come back on how exactly the dependencies are collected.

The descent function is called for each combination of descent value produced by the input e-nodes. For given call we refer these as *current values*. The descent values of an e-node are stored in its *descent cube*, a v-cube attached to the node. The cube has a *target dimension* corresponding to the e-node at hand and one for each of the descent dependencies. Each dimension refer to one v-cube in turn from which the input values are taken.

The number of descent values, the so-called *descent size* may depend on the input values. This can make the entries for the upper indexes in the target dimension void. The size of the target dimension of the attached cube is given by the maximal descent size and as such only known a-posteriori to complete production of the e-tree branch.

2.1.3 Ascent Production

After the descent production of a node and its possible descendants is completed, the node can additionally produce so-called *ascent values* if it defines a *ascent function*. Likewise the descent function, the ascent function can take ancestor nodes as input but it further can depend on successor nodes. In cases where the referred successor defines an ascent function as well, the input refers to its ascent values, otherwise the descent values. For a given call the values from successor node are passed as v-cube. There might other node in between such that the passed cube can have any number of dimensions. The ascent values are stored in the *ascent cube* another v-cube attached to the e-node.

2.1.4 Branching

Within a linear e-tree, an e-node b that depends on another e-node a considers one descent value of a at a time. If you want access all descent values of a , for instance for aggregation purpose, you need a different buildup where a is deployed before b but not as direct ancestor but in a preceding branch in the e-tree, see figure 2.1. The e-tree is deployed in depth-left-first order.

Assume a node has two children a and b where a comes first. For given descent values of the common ancestors of a and b , the subtree rooted in a is deployed first. Then, it is turn to the subtree with head b . From this subtree we can access the all descent values of a and its descendants that where produced under the given decent values of the common ancestors. This can be use for aggregation over the produced values, for instance, we may build the mean over descent values of a , or select an optimal parameter with respect to a certain output.

In the v-cube B where b stores its outputs, the dependency a is represented as singleton *side dimension* and referred cube A is considered *side cube*, indicating that it may hold dimensions not present in v-cube of b s.f. section 2.2.4.

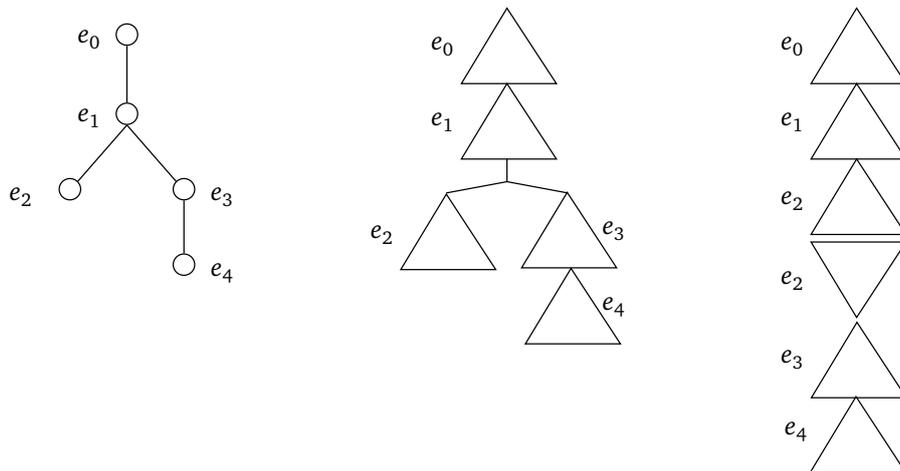


Figure 2.1: A non-linear e-tree with e-nodes e_0, \dots, e_5 . Since e_2 precedes e_3 in depth-left-first traversal of the tree, e_3 can access all descent values of e_2 at a time at least those that are build based on the current descent values of the common ancestors. Alternatively, e_2 can digest the descent values of itself and of possible node below itself by means of ascent production. In either case the result of the subtree under e_2 logically is one value and can be accessed as such by the nodes e_3 and e_4 as sketched on the right.

2.1.5 Partial Orders and E-Trees

Assume a node b that has an ancestor a that produces multiple descend values but on which b does not depend. As the nodes are deployed in the order induced by the tree, b would be repeatedly deployed for any descent values of a . Though it is thinkable to use this in a meaningful way, it would deviate from the intended usage. Instead, the computation is not repeated and the former values are further used.

The cause here is that the linear parts of the e-tree are taken as linear deployment order that is not requested by the actual dependencies. Note however, that the partial order implied by the dependencies would not sufficiently specify the experimental run as we want to allow two nodes to apply changes to the same alterable ancestor value and therefore the deployment order has to be fixed. Besides, we profit from the representation of the experiment as tree of e-nodes as comprehensible outlook.

One can take this as *referential transparency* with respect to the *indexes* of the values that are passed as arguments to the production call. If the production has already been made for the given combination of input indexes the call is skipped. In general referential transparency is a bit delicate notion, as it refers to *value equality* of inputs and outputs of a function, [15]. Here, we refer to the cube-indexes, what makes the concept quiet simple, as it is clear to what equality we refer to.

2.1.6 Production Functions

production functions are those functions of an e-node that understand the argument names as names of e-node and resolve them in the way just explained. They further have in common that they all influence the descent or ascent values of a node. Beside descent and ascent production, they include prune-condition, loop-condition, and initial-sibling-values which are discussed in section 2.3.

2.2 V-Cubes

2.2.1 V-Cube Dimensions

A v-cube has a finite number of *dimensions* that have a unique *dimension name* each. If the v-cube directly stems from a peewit experiment, e.g. holds the output values of an e-nodes there is one dimension for each dependency of the e-nodes plus one dimension for e-node it belongs to. This particular dimension is called *target dimension* and it associated with the *values* the v-cubes holds in a multidimensional *array*.

However, a v-cube is a plain container type and may not or only partially be produced by an peewit experiment. By now the reference of the dimension to e-nodes is only made by the dimension names that may be equal to names of e-nodes.

2.2.2 Recursive Type

Each dimension of a v-cube refers to exactly one v-cube the target dimension of which has the same name as the referring dimension. The target dimension of a v-cube refers to the v-cube at hand whereas the non-target-dimension refer to separate v-cubes each. We refer to the non-target dimension also as *parent dimensions* and the associated v-cubes as *parent v-cubes*, and we speak of the *individual v-cube* when we want to refer to a v-cube excluding its parent cube. So to say, v-cube is a recursive type as it a v-cube includes references to other v-cubes and by now we do not define a non-recursive surrounding container type.

A v-cube may have no parents at all. Still, the data array has at least one dimension since the target-dimension is always present. Figure 2.2 gives an illustration. We demand *referential consistency* in the following sense. If a v-cube B has a parent cube A then dimension of B should also be dimension of A and refer to identical v-cubes.

2.2.3 Dimension Sizes and Values Labels

The number of values the array can hold along a dimension is referred to as *size* of the corresponding v-cube dimension. Referential consistency also applies to sizes: a dimension shared with a parent cube must have the same size in the parent cube and the cube at hand. That is, an individual cube only is responsible for the size of its target dimension.

Given the size of the a dimension we can index the values along that dimension and we assign one *value-label* to each index of the target dimension. Any entry of the array can be identified by a tuple of dimension-name–index pairs. Since parent v-cubes refer to a subset of the dimension at hand, we can associate each entry with a unique *parent value* for each parent v-cube.

Though all dimension of a cube have a fixed sizes, a cube value can be *void* meaning that it is not defined. This has to be covered, however, by a referential consistency constraint: a void entry can only be parent to void entries.

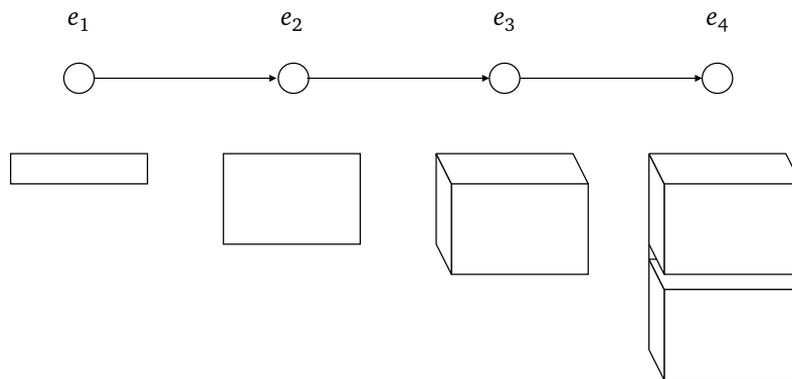


Figure 2.2: Top: four e-nodes. An outgoing edge indicate that the nodes output is input for the other node. Below: Each node is primarily associated with one v-cube. The target-dimension, the v-cube itself, and the respective e-node have the same key, for instance e_4 . For each non-target dimensions the v-cube e_4 references the respective ancestor e-node/v-cube. The v-cube e_4 hence comprises all cubes above it, thereby holding a cascade of arrays with increasing number of dimensions.

2.2.4 Side Cubes

If an e-node b depends on an e-node a from another branch a may have dependencies that b does not have. For v-cubes A and B attached to a and b respectively this results in A being a *side cube* of B . This basically means that referential consistency constrains are relaxed. A might have parent cube to which B does not refer and even refer to different cubes while using identical dimension names as B uses. That corresponds to dependencies of a that are not shared by b and the fact that we allow e-nodes from different branches of the e-tree to have identical names.

The dimension of B referring A is not considered as parent dimension but as *side dimension*. As a side dimension it has size 1 regardless the size of the target dimension of A . One way to look at this is that, from perspective of B , the values of the referred A are v-cubes in turn which enclose the dimension of B not shared with A . This is, however, equivalent to declaring A as side cube in B and thereby allowing it to have dimensions not covered by B .

2.2.5 Consistency Constraints

Each dimension of a v-cube refers to one v-cube. There are three disjoint types of dimensions: target dimension, parent dimensions and side dimensions. We want collect the constraints that a valid v-cube has to fulfill in particular with respect to references to other v-cubes. Assume cube A is a parent cube of B .

- *target name constraint*: If a dimension of B refers to A the dimension name must equal the target name of A .
- *unique size constraint*: If a dimension of B refers to A the sizes of the dimension must equal the length of the label sequence of A .
- *subset constraint*: All parent dimensions of A must be parent dimensions of B .

The two first constraints imply that the dimension of two cubes referring to the same cube are equal in name and size. And by the subset constraint we also have: if A and B have a dimension with equal name then it refers to the same cube.

This is not the case in case A is a side cube of B . In this case from the above constraint only the target name constraint has to be met, along with the

- *singleton side cube constraint*: If A is a side cube of B the size of the corresponding dimension equals one.

Consider an e-node a that depend in ascent on its t-successor b which in turn depends on a for descent. Then, the ascent cube A^a of a has the descent cube B of b as side cube which in turn has the descent cube A^d of a as parent. A^a and B have both a dimension named after a but referring to different cubes. In A^a it refers to A^a itself while in B it refers to the descent cube A^d .

2.3 Advanced Tree Handling

2.3.1 Sibling Values

The descent and ascent production is separately done for each combination of ancestor descent values. In order to compute outputs iteratively we may want to access the previous outputs of the same node, the so-called *sibling values*. Let a be ancestor of b . Then b can access its proper outputs that it made under the previous descent value of a . To this end, the production function of b simply refers to b itself as input node. Since b may have other ancestors we further need to specify the *sibling pivot* a when we actually access a sibling value. Also, we need a function that provides *initial sibling values* for the case that b produces under the first value of a .

2.3.2 Pruning

In case the realization of an e-node and its successors is, for certain input values, logically unwanted or technically impossible, we can pass over the deployment of the node for these input values. To this end the node defines a *prune-condition* that can take the same inputs as the descent function, or a subset thereof, as arguments and outputs whether the tree of values should be pruned at this node. The prune-condition is evaluated before the descent-function is called, e.g. the *pruning* is applied above the node. Unless the prune-condition holds for all inputs, the not computed entries do exist in the value cube but they remain void as it the case when the node has varying descent size.

2.3.3 Looping

E-Nodes may also be used to represent compounds of an algorithm. Algorithms may contain loops where the number of iterations depends on the result of the loops body. So far e-nodes may have varying number of descent values depending on ancestor values that are possible inputs. But since successors, which correspond to a loop body, cannot be accessed as descent inputs, the descent size cannot depend their values. The reason for this restriction is that the descent-function is called before the successors are deployed.

In order to allow *indefinite descent sizes* we can make the node to be deployed repeatedly. The new descent values are appended to those computed so far while the ascent value is overwritten with each iteration. The node defines a *loop-condition* that should output a non-false value until no further iteration is wanted. It can take the same inputs as the ascent functions, including successor nodes that are passed as v-cubes, and it is called right after the ascent-function. For instance, you may check whether the value of a leaf belonging to the current iteration has changed compared the previous one.

2.4 Input Resolution

The descent and ascent values produced by a node are kept in separate v-cubes attached to the respective e-node. The input values for the production are taken from either of the attached cubes of the referred node. This section describes how exactly the inputs are defined. We have to fix several things.

- from which of the attached cubes is the input value taken
- what if there is more than one node that matches the name by which the input is specified
- how are the dimension of the attached cubes set

With those questions cleared we can initialize the attached cubes for each node including the setting of references to the parent cubes.

2.4.1 Node Relations

We first build notions on the relative positions between nodes of an ordered tree. Consider the linear order on all nodes that results from depth-first traversal of the tree respecting the linear orders on siblings. For a given node a this order divides the nodes of the tree into three groups: a itself, the ancestors of a , and the successors of a . More specifically we define the following relations a node b can have to the node a .

- *t-ancestor*: b is on a path from the root to a , including root, excluding a itself
- *s-ancestor*: in depth-first order b comes before a but b is not a t-ancestor of a
- *self*: b is a
- *t-successor*: a is t-ancestor of b
- *r-successor*: all other nodes

The r-successors are the nodes that a cannot access and that is why we do not distinguish further sub-groups.

2.4.2 Input from Descent and Ascent Cubes

When a node is taken as input the actual input values are taken from either from its descent cube or its ascent cube depending on the relation of the referring node to the input node. Basically, the input value is taken from the descent cube if the last production of the input node was a descent.

In terms of the relation of a node b to node a we can fix what value a gets from b value as follows.

- *self*: descent function gets a descent value and ascent function an ascent value.
- *t-ancestors*: a always gets the current descent value.
- *s-ancestor* or *t-successor*: a gets the ascent-values in case b defines an ascent function. Otherwise, a gets the descent-values of b . In either case only those values are passed that are produced for current descent values of the common t-ancestors.

2.4.3 Resolving Input Conflicts

The nodes are linked together via the input names of their production functions. Since the names of the e-nodes do not have to be unique within an e-tree, we need to define how conflicts are. If two nodes have the same name and a third node refers an input by that name the following rules apply.

- Self first.
- Ancestors go before successors.
- Within these two groups the one that comes last in the depth-first order is chosen.

Per e-node there is only one mapping assigning nodes from the e-tree to input names. It applies to all production function of an e-node.

2.4.4 Setting Cube Dimensions

Before we deploy the e-tree the dimension of the descent and ascent-cubes attached to the e-nodes. For each descent-dependency of the node the descent-cube has one dimension (plus one dimension representing the the node at hand), and correspondingly, each ascent dependencies becomes a dimension in the ascent-cube. If a dependency refer to a node that is not an t-ancestor, the dimension is singleton and marked and the referred cube is marked side-cube, which means that certain consistency constraint do not apply.

How do we collect the dependencies in first place? For the dependency resolution we basically have to take the *indirect dependencies* in the wright order into account. To this end a single depth-first traversal suffices. The production function are partitioned into *pre-* and *post-*functions depending on whether they are called before or after the deployment of the child-nodes. The latter group can access t-successors, while the former cannot. If a node e has a dependency a that is used by a pre-function of e , the dependencies of a are added to those of e before the children of e are visited. In case the dependency is due to a post-function, the dependencies of a are added when the children of e are done.

```

[ data(),
  sample(),
  kernel(),
  kernel_tray(),
  [[ c(),
     sub_sample(n_folds=6),
     algorithm(),
     leaf( input_refs =
           [['sample', 'sub_sample']])
    ] ,
  [ c_opt(),
    algorithm( input_refs =
              [['c', 'c_opt']]),
    leaf()
  ]
]
]

```

Figure 2.3: An e-tree with one branching. The e-nodes in the tree are instances of e-node classes such as defined in the code snippet in figure 2.4. The `input_ref` arguments replace input names, such that e-node definition are applicable in either branch.

```

class c(ENode):

    def descent(self):

        return range(-4,4)

class c_opt(ENode):

    def descent(self, leaf):

        vc_opt = leaf.agg([
            ('argmax', 'c'),
            ('mean', 'sub_sample'),
            ('sel', 'leaf', 'accuracy')
        ])

        return [ vc_opt.get_value() ]

class algorithm(ENode):

    def descent(self, data, c,
                kernel, kernel_tray):

        engine = LIBSVM(
            data_tray = data,
            kernel     = kernel,
            C           = 2.**c,
            kt         = kernel_tray,
            )

        return [ engine ]

```

Figure 2.4: Definitions of e-nodes. The node `c_opt` shows the digestion of an s-ancestor by means of v-cube-aggregation. In `algorithm` some library is called.

3 Services

We coded a python prototype named *peewit* [18] that implements the experiment model defined in the previous section and provides various services. Some are for dealing with v-cubes, others, which we refer to as *housekeeping services*, are to support the experimental work-flow. Some services of the latter type only partially rely on the model and could be implemented to some extent within any other top-level framework. Still, they fit the spirit of this work as they show that we can provide helpful code independent of domain types and interfaces. For more detailed information on the services see [17].

3.1 V-Cube Services

V-cube is a sort of data cube [6], a notion for multidimensional container types used in the fields of Online Analytical Processing [4], [14]. As such we can implement various OLAP operations. The presence of sufficient data cube functionality in terms of data manipulation and presentation is crucial for the experimental framework. Due to limited resources we can only realize a fraction of the services that we would like to use. But some are implemented by date and they can illustrate the benefits of cube services: presentation in plain text or as plots, manipulation like selection, aggregation, or merging.

The remaining subsection has three parts. First, we describe concrete services for v-cube manipulation. The last one described, namely merging, leads to the problem of v-cube alignment that we discuss separately. Last, explain how pre-computed v-cubes can be integrated into an experiment.

3.1.1 V-Cube Manipulation

Likewise other data-cubes v-cubes come with several routines of cube manipulation. The most simple ones are the following.

- *value selection*: select or delete a slice given as index or value-label for some dimension
- *permutation of dimension*: change the order on the dimension
- *permutation of values*: for a given dimension change the order on value-labels and the corresponding slices

The manipulations have to account for the main cube as well as for the parent cubes. That's why the next, apparently also simple manipulation, is, in fact, logically a bit more complicated.

- *plain aggregation*: summarize the values along the aggregated dimension into a single value

It will occur that that a parent cube also has the aggregated dimension but the aggregation cannot be applied to its values. Then these values becomes void. The aggregated dimension will be shrunk to a single dummy value. As an example think of taking the mean. This cannot be applied to the parent cube if the parent cube is non-numeric. And even if it were numeric, it would not always make sense to apply mean to it just because we queried the mean values on a node that has the cube as parent.

For aggregations such min or max it makes sense to ask for the/an *fulfilling object* that leads to the result value, for min and max such aggregation are known as argmax and argmin.

- *argument aggregation*: for a set of ancestors dimensions get a combination of values such that the target value holds some condition with respect to all value combinations of those ancestors

Since the main cube comes along its parent cube peewit can easily provide this kind of aggregation as well. The user specifies the *aggregants* that are the dimension over which the aggregation is to be done. The result is a v-cube with has those dimensions of the main cube as proper dimension that are not specified as aggregants. If the user specifies multiple aggregants, the result one v-cube per aggregant.

Another direction of manipulation is *cube merging*.

- *merging*: make a single v-cube out of two

There can be different levels of service here that are defined by different capabilities in matching the dimension and values of the cubes to be merged. At a lower service level, called *straight merge*, the service demands that the cubes are identical in dimension and sizes except the sizes of the dimension along which the cube are to be merged.

3.1.2 V-Cube Alignment

We may also demand more advanced merging that is robust against non-identical names/labels of dimensions/values and tries to solve a matching problem. Assume we are given two cubes ν and $\bar{\nu}$ that may differ in dimension and values. An *v-cube alignment* consists of *matching* of the dimension of ν to those of $\bar{\nu}$ and, for each matched pairs of dimension, of a matching of the indexes of the respective dimensions. Whether the matching are restricted to be total or not depends on the application of the alignment. For instance, with merging partial dimension-matching (in either direction) might be wanted whereas the index-matchings must be total except for the merge dimension.

Certain cases that appear in practice are likely matchable in automatic manner: the index order of a given dimension has changed, the name of a dimension has changed, or in one cube there are additional dimension but with a single index only. For solving the alignment we can analyze name and label information but also compare the content of the cubes, likewise for ontology matching problems.

3.1.3 Descent Values from V-Cubes

Since v-cubes correspond to linear sub-trees we can also apply them as such. Assume you are about defining an e-tree for an experiment that shares dimensions with an v-cube you have at hand. You can add the v-cube like a node into the raw e-tree and when the e-tree is read, this node will be expanded to a linear subtree.

The number of nodes in this sub-tree depends on the ancestors of the v-cube in the raw e-tree. When peewit finds that a dimension of the cube is present as ancestor, this ancestor will be identified with the dimension of the e-node. If no fitting ancestor can be found the dimension and the corresponding parent v-cube translates into an e-node that is included in the e-tree. In the first case peewit also tries to match the descent values of the ancestor node with the values from the parent v-cube such that this linking of v-cubes is likely to work even even if the ancestor produces a smaller number of values than present in the included cube or if the order of values differs.

3.1.4 Precomputed V-Cubes

Another way to reintegrate pre-computed values stored in a v-cube into to the run of an experiment is to attach a (descent and ascent-)*pre-cubes* to e-nodes. With a pre-cube attached the values are from that cube instead of calling the respective production function.

The parent and side cube of the pre-computed e-nodes are attached to the respective e-nodes in the tree that fit to the dimension name.

You can define a pre-cube for an e-node by setting the attributes `pre_dvc` and `pre_avc`.

3.2 Housekeeping Services

3.2.1 Persistent Name Space

By writing experimental code the user introduces names for compounds and values. Peewit extends the live time of the names in the sense that the user can use the names for further coding as well for reading, processing and, presentation of the results. The names of the dimensions of an v-cube are given by the e-name of the node they stem from. For a given dimension, the value labels that are assigned to the positions an entry can take along that dimension or are provided by the user or derived from the values at that position.

The e-names, aka dimension names, are used as follows.

- The names of the arguments of production functions must be e-names. The e-name arguments refer to node in the tree and thereby define the dependencies between nodes.
- This also means that the node names are used in the body of the production function.
- The names of dimension are used in textual or graphic representations of cubes. This way we can easily find the production functions that are responsible for an element in the representation.
- By means of the run-numbers these links remain valid even if the production functions have been changed since the v-cube was produced.

As the concept of value labels is part of the v-cube type the application of label always goes through v-cubes. Value labels correspond to row or column headers in tables and allow identification and selection of rows and columns by names. This can be used to handle results but also in the experimental code itself since v-cube are build during run and can be accessed during run by means of preceding branches or ascent production. Peewit tries to define the labels automatically by means of an heuristic, see [17], but the user can also define the labels explicitly.

3.2.2 Regime on Paths

The way peewit handles paths has three aspects. First, peewit encourages *user space coverage* meaning that all data and code files are accessed within home directory of the user and linked by the project configuration. This facilitates the portability of the files in the sense that it can be synchronized more easily between different machines and allows self-contained project snapshots.

Second, the system uses a simple scheme for naming files and placing them into flat ensemble of directories. Peewit produces several output files such as serialized v-cubes, log-files keeping the standard output, plots, and tex-snippets. The concrete path scheme may be disputable but main profit is that there *some* reasonable scheme that is transparent for the user and readable to the system at the same time.

3.2.3 Archiving

The archiving service aims a more effective persistency of the code and other data that defines an experiments. To this end we employ *version management systems* and add a thin layer upon them which allows us to recover former stage of the covered files in a comfortable way:

- each experiment run gets a *run-number*, a sort of on-top revision number
- the run-number is attached to results and, at low-key, displayed in result representations

-
- with that number at hand you can query peewit to restore the files as they were when the result was produced

Peewit does not conflict with existing version management instances but rather allows their usage at different levels of integration: no archiving, archiving aside possibly existing repos, or archiving that integrates existing repositories. We particularly recommend the service. It does not cause additional work for the user and still gives her a quick access to the code that were used to produce former results. Further, the project can be exported such that other researchers can access past states of the code to comprehend how the results shown for instance in a particular plot were obtained. It presumes, however, that you are ready to disclose your work in such a deep way.

3.2.4 Parallelization

Consider a node e the t-successors of which do not refer sibling values with respect to e or any of its ancestors. In this case the deployment of the subtrees under the different descent values of e are concurrent such that peewit can distribute the computations without further coding provisions.

Once the cluster is setup by means of about ten configuration values it suffices to define the *launch axes* that are the e-nodes over which the computation is to be parallelized. Peewit then cares for distributing the jobs and assembling the partial results and finally returns a single v-cube to the user as if the experiment were run on the local machine.

3.2.5 Timeouts

It occurs that for only certain data sets or parameter values the computation time is much greater than average or, even worse, the computation do not terminate for example due to a flow in an optimization engines.

In order to avoid such hangs you can limit the computation for descent call of a given e-node. If the limit is exceeded the descent computation is dropped and the pruned-ascent function is called as it is done when the prune condition holds.

You can define that time limit by setting the e-nodes `timeout` attribute.

3.3 Future Services

Below are two more housekeeping services that are rather clear on a conceptual level but have not yet been implemented for peewit.

3.3.1 Voluntary Type Checking

Peewit is based on python which is a dynamically typed language. This means that the production function do not have a signature that would explicitly force inputs and output to be of specific types. We do want not step into the *dynamic versus static typing* discussion [11] here, but propose a mechanism for *voluntary type checking* at run-time as follows.

The user can, but does not have to, define e-node methods for type checking, one for each input nodes and one for each production function that make checks on the outputs. The checks are not restricted to test class memberships but can test arbitrary conditions. This means that the user freely decides on whether she wants to check in-depth, superficially, or not at all. If the user defines checking, the checking code is separated from body of the production functions and thereby is out of the way. We might also tell peewit to omit the checks in situations that are not suited for fixing interface agreements.

3.3.2 Progress Indication

When a run takes more time, it is convenient to have an idea what fraction of the computation is done and, ideally, how long the remaining productions will take. This kind of service clearly lies in domain of a top-level experimentation framework and the experiment model of peewit seems well suited for progress indication because we can, for instance, easily count the number of the right-most leaf descent values that have been produced so far.

Unfortunately, it is not that simple to get the total number of descent values before the run it completed since the number of decent values an e-node produces can vary within one run. Though in many cases this number is fixed for all node in the tree it still can be difficult to get that number in advance even having the abstract syntax tree at hand. A simple solution is to provide a progress indication that takes nodes with presently unknown or varying size by means of an unknown factor into account, for instance stating that 65x out of 320x rightmost leafs have been produced.

Bibliography

- [1] D. Albanese, R. Visintainer, S. Merler, S. Riccadonna, G. Jurman, and C. Furlanello. *mlpy: Machine learning python*, 2012.
- [2] M. Braun, S. Sonnenburg, and C. S. Ong. *machine learning open source software*, 2007. <http://mloss.org/>.
- [3] R. E. de Castilho and I. Gurevych. A lightweight framework for reproducible parameter sweeping in retrieval. In C. T. Maristella Agosti, Nicola Ferro, editor, *Proceedings of the 2011 workshop on Data infrastructurEs for supporting information retrieval evaluation*, DESIRE '11, pages 7–10, New York, NY, USA, Oct 2011. ACM.
- [4] P. M. Deshp, J. F. Naughton, K. Ramasamy, A. Shukla, K. Tufte, Y. Zhao, D. Shasha, D. B. Lomet, D. Barbara, and M. Franklin. *Data engineering, 1997. Warning: the year was guessed out of the URL*. 16
- [5] T. Fawcett. PRIE: a system for generating rulelists to maximize ROC performance. *Data Min. Knowl. Discov*, 17(2):207–224, 2008.
- [6] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–54, Mar. 1997. 16
- [7] M. Hanke, Y. Halchenko, P. Sederberg, S. Hanson, J. Haxby, and S. Pollmann. *Pymvpa: a python toolbox for multivariate pattern analysis of fmri data*. *Neuroinformatics*, 7:37–53, 2009. 10.1007/s12021-008-9041-y.
- [8] D. Kroshko. *Openopt*, 2009. <http://mloss.org/software/view/55/>.
- [9] C. Lampert. *chi2 kernel*, 2009. <http://mloss.org/software/view/64/>.
- [10] C. H. Lampert, M. B. Blaschko, and T. Hofmann. Beyond sliding windows: Object localization by efficient subwindow search. In *CVPR*, pages 1–8, 2008.
- [11] E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages, 2004. <http://research.microsoft.com/en-us/um/people/emeijer/Papers/RDL04Meijer.pdf>. 19
- [12] J. M. Mooij. *libDAI: A free and open source C++ library for discrete approximate inference in graphical models*. *Journal of Machine Learning Research*, 11:2169–2173, Aug. 2010.
- [13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and D. E. Scikit-learn: Machine Learning in Python . *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [14] A. Shoshani. Multidimensionality in statistical, OLAP, and scientific databases. In *Multidimensional Databases*, pages 46–68. 2003. 16
- [15] H. Søndergaard and P. Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27:505–517, 1990. 8
- [16] P. Vincent, Y. Bengio, N. Chapados, et al. *Plearn*, 2007. <http://mloss.org/software/view/37/>.

-
- [17] L. Weizsäcker and J. Fürnkranz. Basic instrument for experimental probes in machine learning. Technical Report TUD-KE-2012-01, Knowledge Engineering Group, Technische Universität Darmstadt, Apr. 2012. 16, 18
- [18] L. Weizsaecker. peewit, 2012. <http://mloss.org/software/view/254/>. 16
- [19] M. Zitnik and B. Zupan. nimfa a python library for nonnegative matrix factorization, 2012. <http://mloss.org/software/view/397/>.