Basic Instrument for Experimental Probes in Machine Learning

Technical Report TUD–KE–2012–01

Lorenz Weizsäcker, Johannes Fürnkranz
Knowledge Engineering Group, Technische Universität Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Knowledge
Engineering

Abstract

We propose an instrument based on a abstract experiment model that covers very different set-ups but still yields enough formal grip to allow useful services to be implemented in advance. The model demands the user to entirely decompose the experiments into named compounds. In return the services help with the presentation of results, parallelization on computing grids, or handling of the results containers. In contrast to other machine learning frameworks the tool does not include any algorithms or data-types. Instead it exspect the user to call foreign engines and compounds which it in turn are exempt from providing top-level framework features them-selfs.

Contents

1 Introduction

In the area of experimental machine learning a wide range of frameworks have been developed and published that allow the experimenters to employ approved, high level tools for divers purposes. Roughly speaking, there are two types of framework incentives: the *engine* type wants to solve a specific type of task. In order to employ it the user is expected to build framing code by which the provided engine is called. The *embracing framework* incentive wants that experiments are run through the framework itself which subsequently calls the engines that can be plugged-in in terms of wrappers.

Although writing the embracing code usually is straightforward, it can be error-prune and time consuming as well and it therefore is plausible to share such code too. However, defining an embracing framework is rather ambitious as it somehow has to formalize the intentions of potential users with respect to the experimental setup and the employed data structures. A trade-off between comfort versus generality is inevitable: a narrow specification on what experiments are allows broad pre-build services for the specified type of experiments but it also make the framework less likely to be applicable to the different projects of the users.

This report is about *peewit*, an attempt to build an embracing framework that sets the trade-off much in favor of generality, [18]. It can offer much less services and completely forsakes the user in terms of domain code. There are no engines for preprocessing or training and no domain specific data types. The experiment model that peewit applies only fits to rather simple, *regular* set-ups what limits its application. With respect to the domain, however, it is without much restrain simply because it ignores what the experiment is about.

In order to use the proposed system we first decompose the experiment into components that are called e-nodes. It is not properly specified what an e-node is but it somehow should represent a well-rounded concept within an experiment. The e-nodes represent the dimension of an experiment. The main assumption of the tool is that the experimenter wants to try out various combination of different values for at least some of those dimensions. Typical examples for such experimental dimensions are different datasets, pre-processing, specific parts of the training building point of interest bags, scalar training parameters, or performance measures. The trick of the instrument is, however, that we entirely decompose the experiment into such dimensions whether we want to apply different values to them or not.

If the user provides the e-node decomposition, we can make the system understand to some extend the structure of the experiment. This enables modest but useful pre-build services. The core service is an increased persistence of names: implicit and user provided names of dimension and their values get an extended lifetime in the sense that they can be used for assembling the experiment, but also for querying and presentation of the results.

An rich source for machine learning software is [4] where you can find several embracing python frameworks such as MLPY [1], PyMVPA [7] , scikit-learn [14], or plearn [16] . They have in common that they provide with many training algorithms, pre-processing, and evaluation features such that the user freely can combine those in order to build the desired experiments. In addition to the abundant engines they also provide top- level code that implements typical experimentation pattern like gird-search. These can be used smoothly because that embracing code is aware about the interfaces of the engine shipped within the same framework.

The approach of peewit is different. It does not provide engine code at all nor directly supports the communication between engines by means of defined interfaces since it is not aware of any domain data type. The atomic type trough that code parts pass messages among each other are so-called *values* which are arbitrary objects. The idea is this: the user codes her proper engines and corresponding interfaces, or even better, choses such from well tested libraries like Nonnegative Matrix Factorizion [19], OpenOpt [8], libDAI [13], chi2 kernel [9, 10], Pyriel [6] or one the above mentioned frameworks. The code representing the top-level design of the experiment is then done by means of peewit. The user has control on the top-level design but profits from housekeeping services like container handling, without depending on service code shipped with the applied engines. We can stay with peewit although the domain code is to be changed for the particular demands of the current study. This is important since the cost-gain ratio of a framework becomes better when we can apply it for different occasions. A similar spirit is followed in [5] where also a conservative modeling is proposed but there in conjunction to java.

In the next section we describe the model experiment and thereby introduce notions that are used throughout the report. We then dedicate two section on the services that the instrument provides. The first is about different general service which help in experimental housekeeping and the second focuses on handling and manipulating containers that hold produced results. In a section on future work we propose further services that have not yet been implemented but have rather clear concepts, while a discussions on issues that are not that easy to solve

are put the conclusion section. In the conclusion section we further give list of features that we estimate to be particularly valuable.

This report refers to peewit version 0.7.12 and later but it tries to explain the model and the services in general terms rather than exactly specifying its interface. Therefor this is not an manual. Nonetheless, if we mention a `function` with this font it is an existent function from that version, although these references are mainly for illustration purpose.

2 Experiment Model

2.1 E-Nodes

An *e-node* is a compound of an experiment, that, for instance, is responsible for providing the original data, for specific step in preprocessing, or for realization of the training. There are two motives for putting a part of the experiment code into a proper e-node.

First: we want to probe different variation of the compound or at least keep the compound easily exchangeable. What is typically referred to as a parameter of the experiment is a predestined e-node.

Second: any well rounded compound deserves a proper e-node even though it might be clear that we will not vary the compound within the project. Surrounding the covered code by an e-node can help us to mobilize the code for later projects. Even if do not reuse that specific node in a later experiment, we still have an handy and hopefully well-named template for usage of the covered code.

The two motive seem antagonistic: the first claims an e-node because we want vary a something, the second because want conserve something. But they refer to different levels. The first takes the with-in-project perspective the second to an over-several-projects perspective. Consider for instance an e-node that provides several samples drawn from some input source. The e-node produces different realizations of the compound *sample* and we may take this e-node as blueprint for similar parts in later projects.

Peewit demands that the *entire* experimental code is covered by e-nodes. The experiment is defined by the so-called *e-tree*, an ordered tree of e-nodes. For a start we consider linear e-trees only, e.g. experiments that are defined as a sequence of e-nodes, as in the example given in figure 2.1.

2.1.1 Descent Values

Each e-node produces a list of one or more *descent-values*. This value list may depend on the values produced by an ancestor in the e-tree. Peewit calls the *descent-function* that produces the descent-values for each combination of descent value produced by the ancestors that the descent function takes as input. Therefore, the sequence order of the e-nodes need to comply with these dependencies. If, on the other hand, two nodes do not depend on each-other it does not matter which comes first.

The number descent values, the so-called *descent size* may depend on the input values. For now we want to assume a *fixed descent size* that does not vary over different descent calls and therefor can be associated with the node it-self.

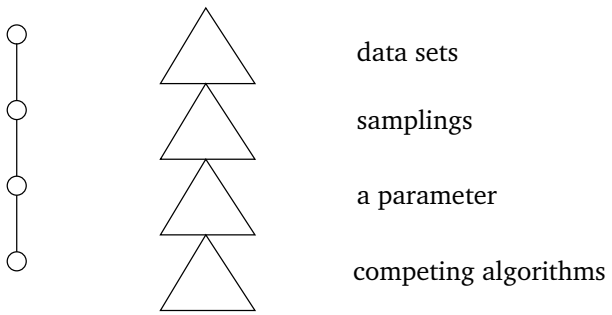

Figure 2.1: A linear e-tree with four nodes (left). Each produces several values per descent call, e.g. per combination of descent values of the ancestors that are taken as inputs. After the run we have a tree of descent value (middle). Assuming fixed descent sizes, the total number of values for a given e-node is given by the product of the descent sizes of that e-node and its ancestors. Therefore, each node corresponds to a multidimensional array of values. See also figure 2.2

2.1.2 Ascent Values

After the descent production of the node and its possible descendants is completed, a node can additionally produce so-called *ascent value*. Opposed to descent the *ascent function* produces a single value only. Likewise descent, the ascent function is called for each combination of ancestor descent values and it therefor may take those as inputs. But it further can depend on successor nodes. In cases where the referred successor node defines an ascent function as well, the input refers to its ascent value. Otherwise all its descent values produced under the current values of the referring node are passed using the container discussed in the up-coming subsection. For reasons that become clear later ascent production is much less important than it it might appear on first sight.

2.1.3 Sibling Values

The descent and ascent production is 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, 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. Since b may have other ancestors we further need to specify the *sibling pivot* a when we actually access the sibling value. Also, we have to provide *initial sibling values* for the case that b produces under the value of a.

Likewise ascent production this mechanism is of minor importance compared to dealing with descent values. The reason here is that you can attach and access arbitrary values to an e-node during production anyway.

2.2 V-Cubes

Assume an e-nodes c that depends on its ancestors a and b and assume a, b and c respectively produce 4, 3 and 2 descent values. The node a finally yields 24 values since the descent function is called $4 \cdot 3$ times each time returning two descent values. Therefore, the descent values of c are stored in an array that has three dimensions, one for each dependency plus one for the node c itself, see figure 2.2. This array along with some additional information is held in in so-called *v-cube*. The v-cube of c has references to the v-cubes of b and c. This way we have the values from a and b based on which the descent values of c were produced at hand. So to say a v-cube actually is a cascade of v-cubes, or one might say v-cube is a recursive type.

For each dimension of the array there is exactly one v-cube which the dimension refers to. The last dimension always refers to the very v-cube it-self, it is therefor called *target dimension* of the cube which in turn is also called *main v-cube* opposed to the *parent v-cubes* that associated with the other dimensions. The target dimension is associated with the node that provided the values entries of the cube. Since the other dimensions refers to other v-cubes they are correspondingly associated with one e-node each. Some of the non-target dimension may not be mentioned as argument of the nodes production function, since indirect dependencies become dimensions as well.

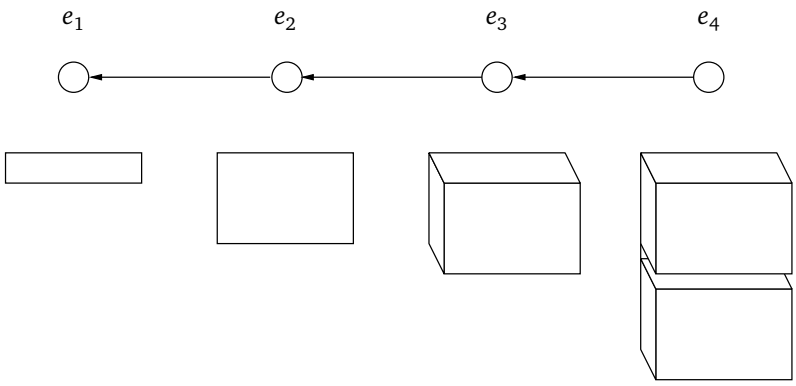

Figure 2.2: For each e-node there is a cube holding its descent-values. The cube has at least one dimension plus one dimension for each ancestor node on which it directly or indirectly depends. The directed edges between the nodes indicate example dependencies.

As stated above, the descent function is not forced to keep the number of output values constant. If the number of descent values varies for some node, the size the cube along that dimension is the maximum descent size over

all descent calls. For ancestor values that lead to a smaller descent sizes, some entries in cube remain void. V-cubes hold the information which entries are actually set and which are not.

Beside the actual values a v-cubes also keeps *value-labels* for its target dimension. Say the array has length k along the target dimension. Then, a list of k label is attached to the v-cubes that fixes how the respective indexes along that dimension should be represented as strings. This reflects the assumed regularity of the cubes content: though the values may differ there is name for i-th entry independent of coordinates along the other dimensions. When this does not make sense we always can fall back to label the positions by the index it-self.

2.3 Non-Linear E-Trees

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 2.3. 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.

Preceding branches, e.g. subtrees that are non-ancestor to an node e are not reflected as dependencies of production of e. One reason is that they always have a single realization given the common ancestors likewise the ascent function return a single value only.

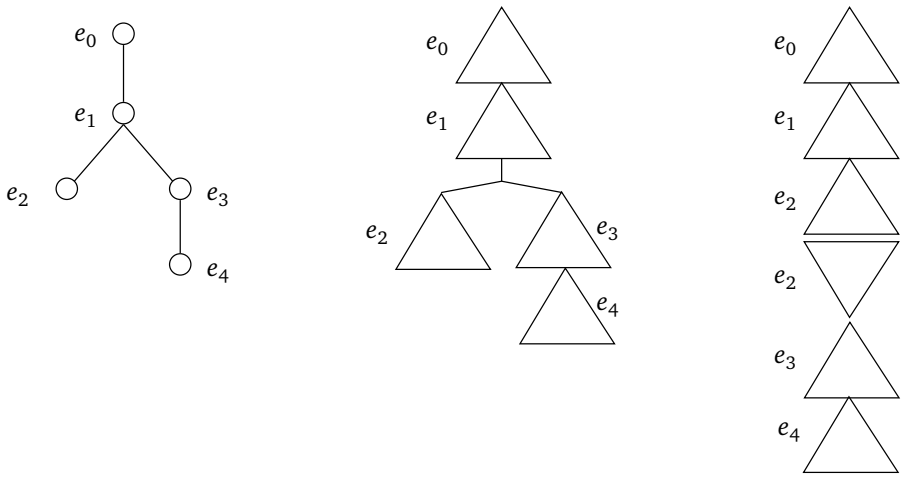

Figure 2.3: Since e_2 precedes e_3 in depth-left-first traversal of the tree, e_3 can access all descent values of e_2 at a time at least those that are build based on the current descent values of the common ancestors. Alternatively, e_2 can digest the descent values of itself and of possible node below 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.

2.4 Advanced Tree Handling

2.4.1 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 inputs. To this end we define the `prune_condition` that can take the same inputs as the descent function, or a subset thereof, as arguments and output 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 exists in the value cube but they remain void as it the case when the descent function has varying output length.

2.4.2 Looping

E-Nodes may also be used to represent compounds of an algorithm. Algorithms often 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 successor, 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 a deployed.

In order to allow *indefinite descent sizes* you 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. You define the function `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.

This gives the concept of e-nodes again another spin since with looping nodes can intrude algorithms more easily.

2.5 Reference Resolution

The nodes are linked together by the names of the arguments of the production functions. This subsection explains how exactly the argument names are resolved to input values.

We start with some definitions 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 the linear 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 subgroups. Also note that descent-production can access ancestors only, ascent-production can additionally access t-successor.

The first to set is to what node a given argument name refers to in case there are two or more nodes with identical e-names. The node is selected as follows.

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

Now we can identify a node with each argument of a production function. Note, that there is only one such mapping per node. For a given node a each e-name refers to the some node b for any referring function of a. The next question is: what values from b are passed to a given the relation of b to a?

- self: a gets its sibling values.
- t-ancestors: a always gets the current descent value.
- s-ancestor or t-successor: a gets the ascent-values in case b defines such. Otherwise, a gets a v-cube holding all descent-values of b produced for the given current descent values of the common t-ancestors.

The notion *current descent values* refers to the fact that the descent function produces a list of possible several values, but the successor nodes only access one at a time. Each combination of ancestor descent values once becomes the current descent values that are passed to the successor production function.

In case a single-valued v-cubes is passed, it is peeled, e.g. the value passed without surrounding v-cube.

2.6 Cube-Dimensions

Before an experiment is run we attach to each e-node one or two v-cubes. The *descent-cube* holding the values produced by the descent function is always attached as all nodes define a descent function. The *ascent-cube* that keep the ascent values is initialized only if the e-node defines an ascent-function.

Each dimension of a cube represents an e-node on the values of which the outputs of the production function and thereby the entries of the cube depend on. For this reason we refer to the dimensions also as *dependencies* of the cube. As indicated above a dependency refers to descent values or to ascent values of the referred node, but only to either of them.

The building of the dependencies works as follows. First, we initialize the *descent dependencies* with the nodes referred by the *descent production functions*. These are all production functions of the node that affect the descent cube. Beside the descent function itself this set contains for instance the `prune_condition`. Correspondingly, we initialize the *ascent dependencies* with the input nodes of *ascent production functions* that have influence on the ascent cube. The notion *production function* means all functions of e-nodes for that the names of argument variables are taken as e-names.

One traversal on the tree suffices for taking *indirect dependencies* into account. 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, that dependency of a is added to those of e before the children of e are visited. If a has a dependencies used by a post-function of e, it is added to the dependencies of e after the children of e are done.

An necessary condition for any node a to be added as dependency of a node e is that a is t-ancestor of e. If a is s-ancestor to e, the production functions of e may access values from a and from the subtree below, but those nodes do not become dimensions in es cube. Nonetheless the dependencies of a that are t-ancestors to e become dimensions of es cube since its entries indirectly depend on them. If we want to refer to an s-ancestor as cube dimension, we have to define a proxy node that take its values and is t-ancestor to e.

Peewit presumes *referential transparency* in the following sense. When the descent function is to be called a second time under the same combination of *indexes* of the ancestor descent values, the call is skipped, since the values for that inputs has already been computed. This occurs whenever there is node e that has an ancestor a that produces multiple descend values but on which e does not depend. By catching this the dependencies are reflected in production calls. In general referential transparency is a bit delicate notion, as is refers to *value equality* of inputs and outputs of a function, [15]. Here, in contrast, we refer to the cube-indexes, what makes the concept quiet simple, as it is clear to what equality we refer to.

3 General Services

This section we explain some housekeeping services, e.g. particular features that help to lessen the banal struggles in realizing experiments. Some of these servicess do not rely directly on the experiment model and could similarily be implemented within any other top-level experiment framework.

3.1 Paths

The way peewit handles paths has three aspects that we want to discuss. First, peewit encourages *user space coverage* meaning that all data and code files are accessed within home directory of the user. Second, the names of output files are structured revealing some meta-information to the user as well as to the system. Third, peewit uses a small, flat directory structure in order to put each file in its place.

3.1.1 User Space Coverage

When you start a project the first thing to do is to fix a *base directory* relative to which all paths are defined, let it be the path to the *project directory* under which any output files will be places, let it be the path to a local copy of some data repository. This base directory in turn is given relative to the users *home directory*. The idea is that all dependencies beside that on peewit itself are part of the project configuration what allows peewit to transport and snapshot the project thoroughly.

3.1.2 Filenames

Peewit produces several output files such as serialized v-cubes, log-files keeping the standard output, plots, and tex-snippets. These kinds of files have different suffices and are stored in separated directories, but for given run but they have a common *base-name*. This base-name consist of four parts separeted by double dashes.

```
module--version--target_ename--part
```

Each part has a meaning, so to say the file-names contain meta-information that can be understood by the user. The meaning of the name parts are the following.

- `module`: name of the python-file from which the experiment run is called
- `version`: some string given by user to specify the concrete version of the experiment
- `target_ename`: the name of the target dimension of the v-cube return by the run-call
- `part`: the computation may be split into several parts, this name part indicates to what part the file belongs to

To allow peewit to parse the filename as well, you should avoid any names containing double dashes or double underscore.

3.1.3 Directories

The second compound of the file regime is a flat ensemble of directories rooted by the *project directory*. You find therein the following sub-directories

- `vcubes/`: V-cubes are saved and searched here. There additionally searched in `kept/`.
- `logs/`: Std-out and err-out of your code are written to files in this directory.
- `plots/`: Opposed to `vcubes` subdirectories are created here in order to keep the plots of different experiment in a proper place each.

- `.peewit/`: This directory is mainly for internal usage. It keeps information that is needed during parallelized runs, or for archiving purpose. Occasionally, the user may want to look into it or even make changes, for instance, to overwrite the base directory when it is an imported project.

There is no intended place for the project code. You can either place the modules directly into the project directory or create some sub-directory to hold the code.

3.2 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

Version management systems store the version information on tracked files in a *management directory* such as `dir/.hg/`, `dir/.git/`, `dir/.svn` where `dir` is the root directory of the local copy of the repo. Peewit uses a proper management directory with special path such that the version information hold by peewit will not interfere with possible other version information on the same files initiated by the user.

Peewit keeps track on a set of *covered directories*

- *external repositories* that exist independently of peewit can also be covered
  - *passive integration*: peewit only keeps track of the current revision
  - *active integration*: peewit also commits to the repo before each run
- *internal repositories* are the repos that peewit manages itself on
  - the project directory excluding all sorts peewit generated files such as logs and v-cubes-files
  - directories that are registered python paths for peewit
  - further directories of your choise

The internal repos are tracked via git and the managing directory is `project_dir/.peewit/.git_peewit` such the covered files can also belong to other repos you use. In case you want the run-number refers to the 'official' revision numbers of an existing repo such as a code bases from external software projects, you can register the local copy as an external repo. For instance peewit itself is by default treated as external repo with passive integration such that you can run former experiments even though you now work with updated version of peewit.

In case of passive integration of external repos the run-number refers to the revision that might differ from actual state of files used for the experiment. In this case peewit will give a warning but if you made substantial local changes the experiment will not be reproducible. In order to avoid such gaps you have to commit your changes by hand before launching a run.

We particularly recommend the service. It does cause not additional work for the user and still gives her a quick access to the code used for producing results in the past.

You also can export the project to single tar-file using the function `export_project`. If you have used the archiving service, this export includes the entire version history. Therefore it is thinkable to partially drop details in the experiment documentation and reference an online-available export of the project instead. Due to run-numbers and the decomposition into e-nodes other researchers have good changes to be able to get an answer to questions on a particular aspect of an experiment without further guidance. Making full export abailable presumes, of course, that you are ready to disclose your code in such a deep way.

3.3 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.

The user has to set the connection-data for the computing *cluster* and set the *launch axes* that are the e-nodes over which the computation is to be parallelized. For technical reasons an axis needs to have fixed descent sizes and the user may additionally have to provide that number in case peewit determine it read from the code. 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.

Nonetheless, the workflow is a bit different than without parallelization. We you *launch* an experiment on a cluster the call of the main experiment module terminates after the job are distributed. In order to collect the result you call the module again. If some jobs have completed and other have not, peewit tries to collect and return the results available so far.

3.4 Persistent Name Space

By writing experimental code the user introduces names for compounds and values. Peewit extends names live time in the sense the user give to a thing once and then can use the name for further coding as well for reading, processing and, at least for internal use, presenting results.

The dimensions of an v-cube given by the e-name of node they stemm from. Each dimension has one label for each positions an entry can take along that dimension and the label is derived from the values at that position.

Both, e-names and lables, are strings and both are explicitly or implicitly defined by the user.

The e-names, aka dimension names, are given by the class name of the e-nodes and are subsequently 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 in the body of the production function the input values are referred by the name of the node they stem from.
- 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 archiving services run-number these links remain valid even if the production function 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 when the user does provide labels herself. The heuristic in doing so works as follows.

- If the values come with a `get_name()` function, the returned names are use as labels.
- Otherwise peewit checks whether all values are returned by means of local variables and if so, takes the names of the the variables as labels.
- If the neither of the above methods can be applied, the labels are guessed directly base on the value. For instance numeric values can be directly transformed to labels.
- Since the labels are sorts of headers, they are fixed for a given node and cannot depend on values of parent nodes. In case they d, the labels are replaced by their indexes.

The idea for using local variable from the function body as labels is this: speaking names for variable are advisable in any case as they help reading the code. And once they are chosen well, it would be a wast not to use them further.

4 V-Cube Services

The nodes produce values that are kept in v-cubes. The v-cubes can be stored, loaded, displayed in on form or the other, manipulated, and reused in definitions of further experiments.

4.1 Saving and Loading

A v-cube has a `save()` routine that tries to store it under the its base-name, see 3.1.2, where it inherits the name-parts `module`, `version`, and `part` from the producing e-tree. Provided that the stored v-cube is in the directory `vcubes/` or `kept/` it can be loaded by passing the base-names to v-cube-constructor. But you can also pass the path to the stored cube directly and independently of its locations.

When we store a v-cube is have to assume that the values of the main cube as well as the values of the parent v-cubes can be serialized. If one of the cubes contains values that cannot be serialized all values of that cube are replaced by a string representation.

4.2 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.

4.3 Representation

There are different ways to output the values of an v-cube.

The *plain-text output* has two parts. In the first part all dimension are listed along with their value-labels. This is somehow the header of the text representations. In the second part the cube-values are given in the nested list representation of arrays as is it used by numpy [3]. Of course this only makes sense if the values have meaningful text representations.

In case values are scalar we also query peewit to present the values in *generated plots*. To this end the user fixes which dimension is the x-axis and which dimension should be coded into different line/dot colors. For remaining dimension with more than one values peewit accounts by drawing multiple plots and putting the information of the respective values into the titles.

4.4 Manipulation

Likewise other data-cubes v-cubes come with several routines of cube manipulation. The more 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. Thats is why the next, appearently 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 merger demands that the cubes are identical up to one dimension. At higher level the merger is robust against non-identical names/labels of dimensions/values and finds a matching on its own considering the order on dimension and values. The implementation of the higher levels has been started but will remain subject to continuoes improvements as it solves a sequence matching problem.

5 Future Work

5.1 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. It is quite complex and greatly driven by personal taste. Still, we drop two arguments, one in favor and against dynamic typing.

Peewit mainly addresses non-experienced coders and for those we believe dynamic typing to be more suited simply because the number of language elements needed to make the code run is smaller. This better fits the spirit of peewit which wants allow non-wizard programmers to quickly evaluate an hypothesis on the behavior of some algorithm or data set. On the other hand, static typing can also save time: when a function is called with the wrong inputs types this is indicated right away and not from somewhere in the body of the function, possibly several stack frames below.

We plan to add the following mechanism that supports type checking at run-time. The user can define methods for type checking, one for all input nodes and one for each production function makes checks on the outputs. Which of the she define is left to her. Also, the checks are not restricted to test class memberships but can apply arbitrary conditions.

An advantage of this mechanism is that the user freely decides on whether she wants to check superficially, in-depth, or not at all. If she defines checking, the checking code is separated from body of the production function it refers to and thereby is out of the way. She can even tell peewit to omit the checks in situations that are not suited for considerations on interface agreements.

5.2 Progress Indication

When a run takes longer 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 total number of descent values before the run it completed since the number of decent values an e-node produce can vary within one experiment 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.

There are different approaches to deal with that all assuming a fixed number of descent values. One could make a trial run that runs the experiments until the descent production of each node has been called once and thereby reveal the number of output values. But this trial run can take some time, in particular in case where much load lies on earlier branches. Alternatively, one could simply start the progress indication not until the descent numbers are collected. However, run-time estimation are particularly demanded right after the start.

Since we are rarely interested in the absolute number of produced values but rather relative progress, we might indicate the number obtained versus grand total by means of unknowns: $7x$ out of $30x$. We confine ourself with the available information and assign the unknown to the term x.

5.3 Table Snippets

In section 4.3 we mentioned two ways to output cube value, as plain-text and as plots. We plan to re-add outputs as latex-table-snippets as we had implemented it in an earlier version of peewit. For publications such snippets might be too generic, but they still could serve as draft that is further edited by hand. For preliminary reports the produced table will in many cases be neat enough and the snippet files may directly be included in a latex document. This means that you only have to call the export again and possibly change the referring filename in order to update the report with a newer version of the results.

With 2d-plots as implemented by now, we have one *base dimension*, the x-axis, and one *extra dimension* the colors of the lines. Together they represent the two *inner dimensions* that are absorbed by a single *output object*,

namely one plot. In case the v-cube has more non-singleton dimension, these become the *outer dimensions* which are followed by multiple plots. Likewise, tables have two base dimensions, the rows and columns, and an indefinite number of extra dimension given by groups, super-groups, and so on, of rows and columns that are separated by extra lines in the table.

Each group can be titled by the respective value label. But here comes a technical difficulty: the width of columns often is rather small such that we have to abbreviate the column title. This means that we have to provide an automatic abbreviation mechanism and/or also give the user the opportunity to fix short forms herself.

6 Conclusion

The proposed instrument is intended to offer quick realization and result examination of straight-lined empirical studies varying multiple experimental compounds. The tool is agnostic to area under studied by that it does not take data types or algorithm into account. Instead it tries to take struggle from the realization of empircal investigations by caring for everyday problems that decelerate the evidence to be available in a tidy form.

While writing the tool we designed a simple experiment model. An experiment is a tree of e-nodes each of which branches along its values. After deployment linear subtrees starting at the root can be represented as v-cubes which in turn can be stored and further processed. For a sound evaluation of this model including ascent production, sibling value access, and looping, we lake sufficient amount of application cases. But we have already employed the tool during its development on a research project, though an earlier version with different core design [17]. In the following we describe positive experience we made in practice and subsequently discuss issues for which we do not yet know how they should be tackled.

6.1 Highlights

Here are some points where peewit can, according to our subjective experience, save time and effort.

File Regime

Peewit does not solve the problem of naming and later finding different versions of an experiment. This issue is defused in two ways: first, by taking the name of main module as name part of output file, and second, by providing a simple systematics on where to keep the files.

Run Numbers

We almost always turned the archiving service on. It cost small delay when starting a run and we only used the restore function a couple of times. But it is very agreeable that we do not have to make exhaustive notes on each run because we easily look up the details of some dimensions by passing the run-number attached to a plot to the restore function.

Grid Computing

Parallelization of fully concurrent computations is logically straight forward. Still, in practice you have to solve a bundle of issues: formulation of the job-assignments, transferring file to and from the cluster, and merging the partial results. With peewit you set about a douzen configuration values, most of them independent of the current project, specify the concurrent dimensions, and then run an experiment very much as you do on your local machine. Though you still may end up in trouble shouting, you will much less need to acquire knowledge on grid computing that has little to do with your interest in, say, word disambiguation or change point detection on audio data.

Experiment Outline

An e-tree defines an experiment. The user provide the structure of that tree by stating the *raw e-tree* in *branch notation* that particularly readable if the tree has a view branches only as it is the case for most experiments. This explicit tree representation also serves as outline of the experiment much like an index of contents of an article or book. It does not disclose the full information on the semantics and dependencies of the nodes but with speaking node names, likewise the section titles in the table of contents, you get a pretty good idea on the design of the experiment.

6.2 Issues

The are, of course, also many issues that peewit does not solve or even newly introduce and for some it is not clear whether the design of the framework impairs or rather defuses them. In the remainder we discuss a selection of such.

Dependency Outline

Remind the last highlight, *experiment outline*, from the previous subsection. When looking on the tree with an editor of a reasonable IDE, for instance [2], we can click on node and the IDE will open its definition. This way we can easily look up the details on a node and see, for instance, what the exact dependencies are. There is, however, no outline that shows all the dependencies as given by the graphical interface of other experimentation such as [12].

Structuring the Code

It is tempting to see in the decomposition of the code compounds into nodes a mean for effective reuse of compounds. But if we want to recycle e-nodes, we also need some systematics for keeping and finding them. In this sense peewit does not provide assistance in structuring the code. Things even get worse by the fact that e-nodes are very heterogeneous with respect to their inner complexity, e.g. the number code lines they contain, their outer complexity, the number of implemented production functions, and their dependencies, but also with respect to their scope: some are of interest for specific experiment only, other might be used in several projects.

If you approach the maintenance of your code base in airy manner the following is likely to happen. You define some nodes and use them in an experiment. Then in another experiment you define other e-nodes but also import nodes from the former one. You will also redefine nodes and end up in confusion because some node have a lasting semantic and some are present in different variants.

Partially, this can be mitigated by parameterizing the nodes. You can define an `init`-method that set an parameter to which you can assign values in the raw-tree. This way you do not have to introduce another node for the parameter. Alternatively, input arguments of production functions can also be optional, such that the values can be taken from ancestors nodes but do not have to.

Deviant Inheritance

E-Nodes and v-cubes are python-classes. However, both stem from a base-class `LeanDescendant` that alters to some extend the inheritance mechanism as defined in the python language. The motivation is to allow simpler definition of e-nodes, as well as of wrappers around libraries that can be called in the production functions. The inheritance processing deviate as follows.

- The constructor is `init()` instead of `__init__()`.
- The constructors of the super-classes stemming from `LeanDescendant` are called automatically in `mro`-order.
- Key-Word arguments of for the super-classes can also be use by the descendant.

We like these alternation and believe they are profitable for small artifacts as the top-level definition of machine learning experiments. But they might be confusing for people how are used to standard python inheritance. On the other hand, the user only has to deal with it in case of e-nodes and some classes shipped with projects exmaples. In case of e-nodes the user can switch the deviant inheritance off by overwriting `__init__()`.

Bibliography

[1] D. Albanese, R. Visintainer, S. Merler, S. Riccadonna, G. Jurman, and C. Furlanello. mlpy: Machine learning python, 2012. 3

[2] Aptana Studio and Titanium Studio. Pydev 2.4.0. `pydev.org/`, 2011. 18

[3] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, and T. Oliphant. An open source project numerical python david ascher paul F. dubois konrad hinsen jim hugunin travis oliphant with contributions from the numerical python community., Sept. 07 2001. 13

[4] M. Braun, S. Sonnenburg, and C. S. Ong. machine leanring open source software, 2007. `http://mloss.org/`. 3

[5] 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. 3

[6] T. Fawcett. PRIE: a system for generating rulelists to maximize ROC performance. *Data Min. Knowl. Discov*, 17(2):207–224, 2008. 3

[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. 3

[8] D. Kroshko. Openopt, 2009. `http://mloss.org/software/view/55/`. 3

[9] C. Lampert. chi2 kernel, 2009. `http://mloss.org/software/view/64/`. 3

[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. 3

[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`. 15

[12] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. Yale: Rapid prototyping for complex data mining tasks. In L. Ungar, M. Craven, D. Gunopulos, and T. Eliassi-Rad, editors, *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 935–940, New York, NY, USA, August 2006. ACM. 18

[13] 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. 3

[14] 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. 3

[15] H. Søndergaard and P. Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27:505–517, 1990. 9

[16] P. Vincent, Y. Bengio, N. Chapados, et al. Plearn, 2007. `http://mloss.org/software/view/37/`. 3

[17] L. Weizsäcker. Persistent name spaces for uniform experiments. Technical Report TUD-KE-2010-04, TU Darmstadt, Knowledge Engineering Group, Apr. 2010. 17

[18] L. Weizsaecker. peewit, 2012. `http://mloss.org/software/view/254/`. 3

[19] M. Zitnik and B. Zupan. nimfa a python library for nonnegative matrix factorization, 2012. `http://mloss.org/software/view/397/`. 3