

University of Belgrade  
Faculty of Organizational Sciences  
Center for Business Decision Making

# WHIBO Complete user guide

Belgrade, November 2013

# Contents

|  |    |
|--|----|
| Introduction.....  | 1  |
| Black-box approach .....   | 1  |
| White-Box approach .....   | 2  |
| WhiBo component repository and Generic decision tree (GDT) algorithm ..... | 2  |
| Sub-problems and solutions (reusable components) .....                     | 3  |
| Generic decision tree (GDT) structure.....                                 | 4  |
| Installation guide .....   | 5  |
| WHIBO environment.....   | 8  |
| WHIBO generic decision tree (GDT) operator GUI manual.....                 | 9  |
| WHIBO generic decision tree (GDT) evolutionary search operator GUI manual  | 10 |
| WHIBO testing environment manual .....                                     | 14 |
| Application examples.....  | 16 |
| White-box component based design and application .....                     | 17 |
| Recreation of well-known algorithms with component based approach .....    | 18 |
| CART algorithm.....  | 18 |
| C4.5 algorithm .....   | 22 |
| CHAID algorithm .....  | 27 |
| Modifying generic decision tree algorithms.....                            | 32 |
| Modifying parameters .....   | 34 |
| Replacing components for sub-problem .....                                 | 35 |
| Adding new sub-problem and component to an existing algorithm .....        | 36 |
| Generic decision tree evolutionary search design and application .....     | 39 |
| Modifying algorithm search space .....                                     | 47 |
| Modifying parameters .....   | 48 |
| Replacing components for sub-problem .....                                 | 49 |
| Adding new sub-problem and component to an existing algorithm .....        | 49 |

|                                  |    |
|----------------------------------|----|
| Extending WHIBO .....            | 52 |
| Developer guide.....             | 55 |
| Appendix A .....                 | 64 |
| ID3 algorithm .....              | 64 |
| CART algorithm .....             | 64 |
| C4.5 algorithm .....             | 64 |
| CHAID algorithm .....            | 65 |
| Distance measure .....           | 65 |
| Appendix B.....                  | 65 |
| Subproblem: Create split .....   | 65 |
| Subproblem: Evaluate split ..... | 69 |
| Subproblem: Stop criteria .....  | 71 |
| Subproblem: Prune tree .....     | 72 |
| References.....                  | 73 |

## Introduction

WhiBo is a RapidMiner (Mierswa et al. 2006) plug-in for “white-box” component based design of decision tree algorithms for classification and evaluation of these algorithms and their parts. It is intended to be used by typical end users, research scientists and algorithm developers. The main idea of WhiBo is to offer standardized components for algorithm design which will enable simple design and performance testing, easy extension of the component repository and creation of new generic algorithms. Currently, WhiBo provides one generic algorithm, a graphical interface and a component repository for design of decision trees for classification. A framework for performance testing is implemented in WhiBo as well. WhBo plug-in and source code, is available from [www.whibo.fon.rs](http://www.whibo.fon.rs). Source code is documented thoroughly and accessible from the web site through the API documentation. The web site also provides installation guide and number of tutorials for end users, algorithm developers, and research scientists.

## Black-box approach

Data mining algorithms are usually implemented in a “black-box” manner. This means that the user defines input data and parameters (if needed) for the algorithm, and the algorithm produces a model. The user has no other possibilities to modify the algorithm to better adjust to data. The “black-box” approach is satisfying for most users. On the other hand, implementation of algorithms as a “black-box” makes it more difficult for algorithm designers who want to use parts of the existing algorithm to create new algorithms. The structure of black box algorithms demands reimplementing of algorithms and their parts from the scratch. “Black-box” implemented algorithms are harder to evaluate and analyze, because it is not clear which part of the algorithm has influence on overall algorithm performance.

## White-Box approach

The “white-box” approach allows the user to define parameters, and inputs (as in black-box algorithms) of an algorithm, but also the building blocks (i.e. components) of the algorithm. These components are solutions for typical sub-problems consistently encountered in the process of constructing the appropriate model for the data at hand. This way, algorithmic solution becomes more data and user driven, since it enables the users to intelligently select components of the algorithm which best address the problems of the specific data. Moreover, good ideas from algorithms are saved within components, so they can be used in other algorithms.

White-box approach offers several advances in comparison with black box algorithms (Sonnenburg et al, 2007).

- Combining advantages of various algorithms,
- Comparing algorithms in more details,
- Building on existing resources with less re-implementation,
- Easier “bug” detection on the level of components,
- Collaborative emergence of standards.

## WhiBo component repository and Generic decision tree (GDT) algorithm

WhiBo includes a reusable component repository for design of decision tree algorithms. These components were extracted from “black-box” algorithms:

- ID3 (Quinlan JR, 1986),
- C4.5 (Quinlan JR, 1993),
- CART (Breiman et al, 1984),
- CHAID (Kass GV, 1980)

and improvements (distance measure identified in (Mantaras, 1991). Description of analyzed algorithms and partial improvements could be found in Appendix A.

## Sub-problems and solutions (reusable components)

In WhiBo algorithms are built by choosing building blocks (i.e. reusable components - RCs) for each sub-problem. The problem of building decision tree model is divided into sub-problems that are generalized algorithm structures with the same input and output structure identified in all analyzed algorithms. Every sub-problem with defined inputs and outputs can be solved in many ways, i.e. with various a reusable components (RCs). That means that every RC solves a specific sub-problem which has the same I/O.

Table 1 shows identified sub-problems and components with their corresponding I/O that are currently implemented in WhiBo.

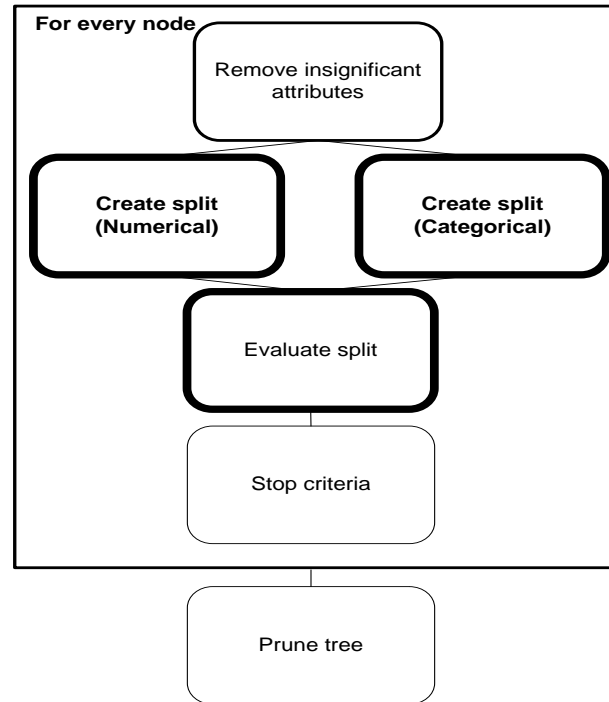
| Sub-problem                     | Reusable component  | Input                   | Output  |
|---------------------------------|---|-------------------------|---|
| Remove insignificant attributes | F TEST (numerical attributes)<br>CHI SQUARE TEST (categorical attributes) | Dataset in current node | Dataset in current node (reduced)               |
| Create split (Numerical)        | BINARY  | Dataset in current node | A split candidate                               |
| Create split (Categorical)      | BINARY  |                         |   |
|                                 | MULTIWAY<br>SIGNIFICANT   |                         |   |
| Evaluate split                  | CHI SQUARE  | A split candidate       | The best split in current node                  |
|                                 | INFORMATION GAIN  |                         |   |
|                                 | GAIN RATIO  |                         |   |
|                                 | GINI<br>DISTANCE MEASURE  |                         |   |
| Stop criteria                   | MAXIMAL TREE DEPTH  | Current tree model      | Signal for stopping tree growth in current node |
|                                 | MINIMAL NODE SIZE   |                         |   |
| Prune tree                      | PESSIMISTIC ERROR PRUNING (PEP)   | Current tree model      | Pruned tree model                               |
|                                 | MIN LEAF SIZE (MLS)   |                         |   |

**Table 1 - Sub-problems, reusable components with standardized I/O for Generic decision tree algorithm**

Sub-problems and reusable components implemented in Whibo are described according to Tracz (1990) in Appendix B.

## Generic decision tree (GDT) structure

The GDT structure proposed in WhiBo is shown on Figure 1. For sub-problems that are bolded it is necessary to define a sub-problem, while for other sub-problems RCs are optional to use. “Create split” (numerical, and categorical) and “Evaluate split” RCs are necessary for decision tree growth. Besides that, there are no restrictions for combinations of RCs.



**Figure 1 - Generic decision tree (GDT) algorithm**

The proposed GDT structure and component repository enables:

- Reconstruction of the original algorithms in the parts that were analyzed.
- Creation of hybrid algorithms with components.
- Extension of the component repository by analyzing new algorithms or partial improvements which can be incorporated in sub-problems with the same input-output structure.
- Definition of new sub-problems which can be incorporated in GDT structure.

## Installation guide

There are two ways of plug-in installation. First way is over RapidMiner marketplace, and is done in following steps:

1. Open **RapidMiner**.
2. Press **Help->Updates and Extensions (Marketplace)**.
3. Enter WhiBo as search term.
4. Click **Search** button.
5. Select WhiBo extension for installation (or update).
6. Press **Install x packages** button.

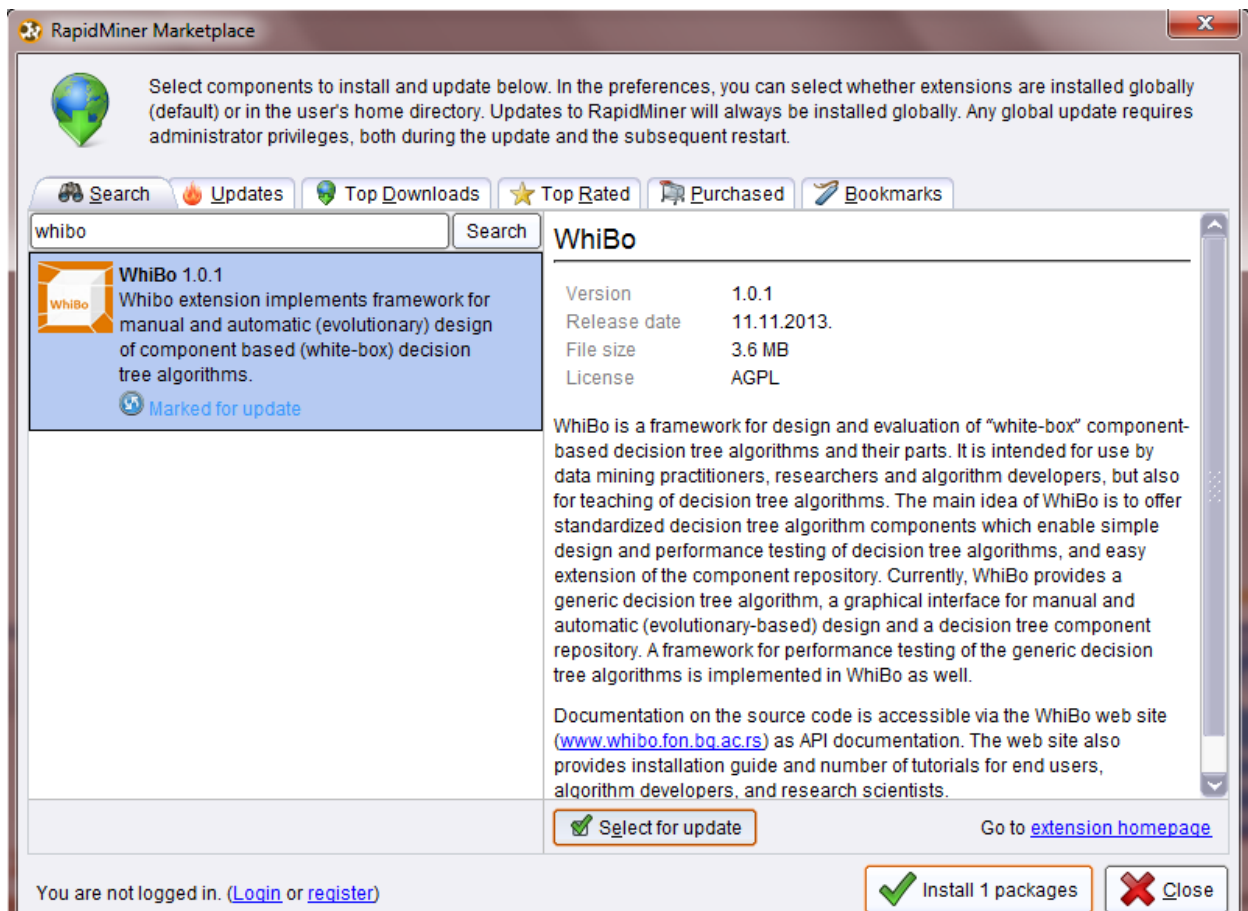


Figure 2 – RapidMiner Marketplace

7. Read and accept the terms of license.
8. Press **Install x packages** button.



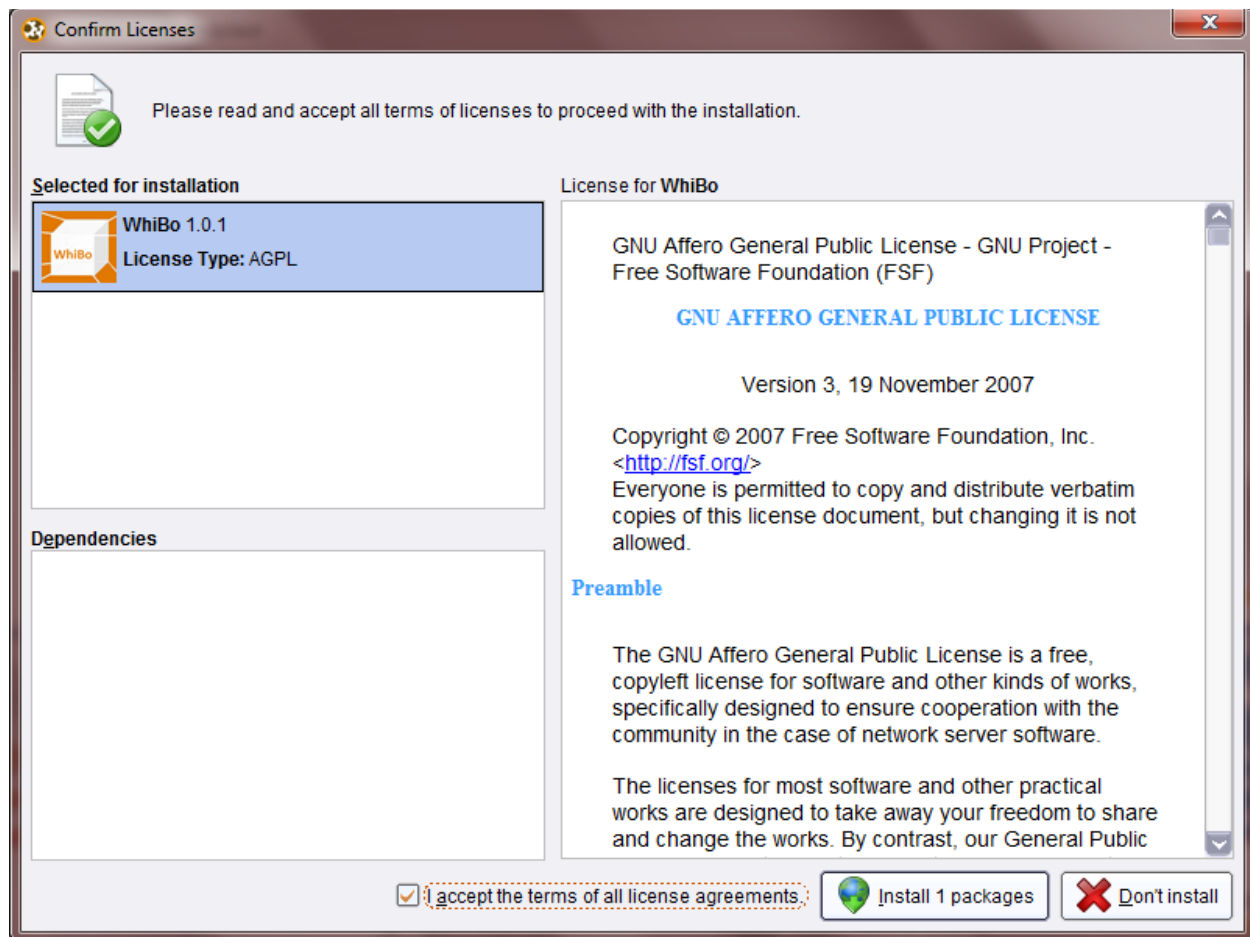


Figure 3 – Confirm license dialog

RapidMiner will install plugin and restart in order to apply changes.

Second way: When RapidMiner is downloaded, installation of WhiBo can be completed in two simple steps.

1. Download WhiBo.jar archive from download section of WhiBo page <http://www.whibo.fon.rs>.
2. Place WhiBo.jar file in Rapid Miner's plugin-folder :  
...Program Files\Rapid-I\RapidMiner\lib\plugins

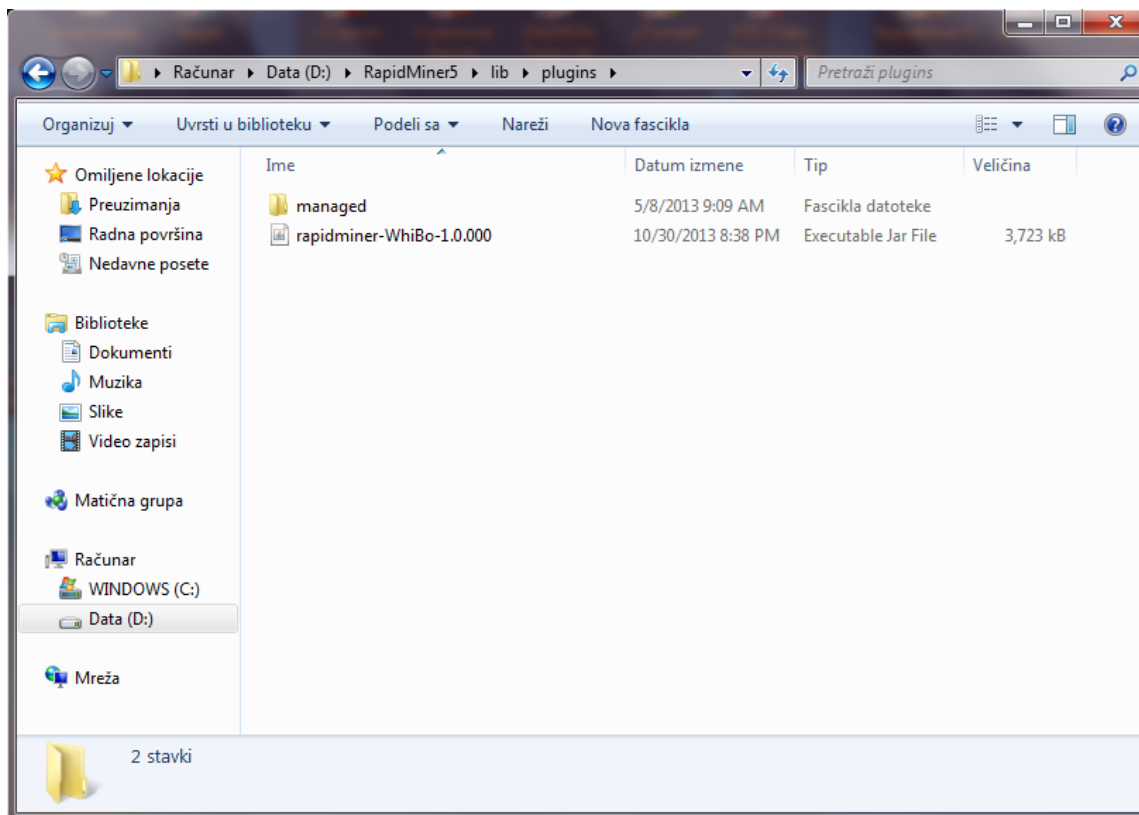


Figure 4 - Placing WhiBo plug-in

If both steps were done correctly user can start the script by double clicking, and RapidMiner with WhiBo environment will start.

## WHIBO environment

WhiBo environment currently implements two operator groups:

- Trees – contains *Generic decision tree operator* and *WhiBoGDT Evolutionary Search operator*.
- Validation – contains *Custom cross validation with log* and *Significance 5X2 cross validation F-test operators*.

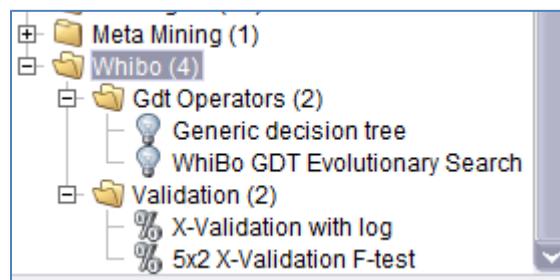


Figure 5 - WhiBo operator group

## WHIBO generic decision tree (GDT) operator GUI manual

WhiBo generic decision tree user interface contains four panels:

Left panel contains an array of buttons. Every button represents a concrete sub-problem for a decision-tree algorithm design.

Central panel contains:

- Available RCs of selected sub-problem from the left panel.
- Available parameters (if available) for selected RCs.
- Buttons for including or disabling a RC from the current decision tree structure.

Right panel shows current state of user designed algorithm (saved sub-problems, RCs and parameters).

Top panel contains options for creating new, saving current or opening existing generic decision tree algorithm.

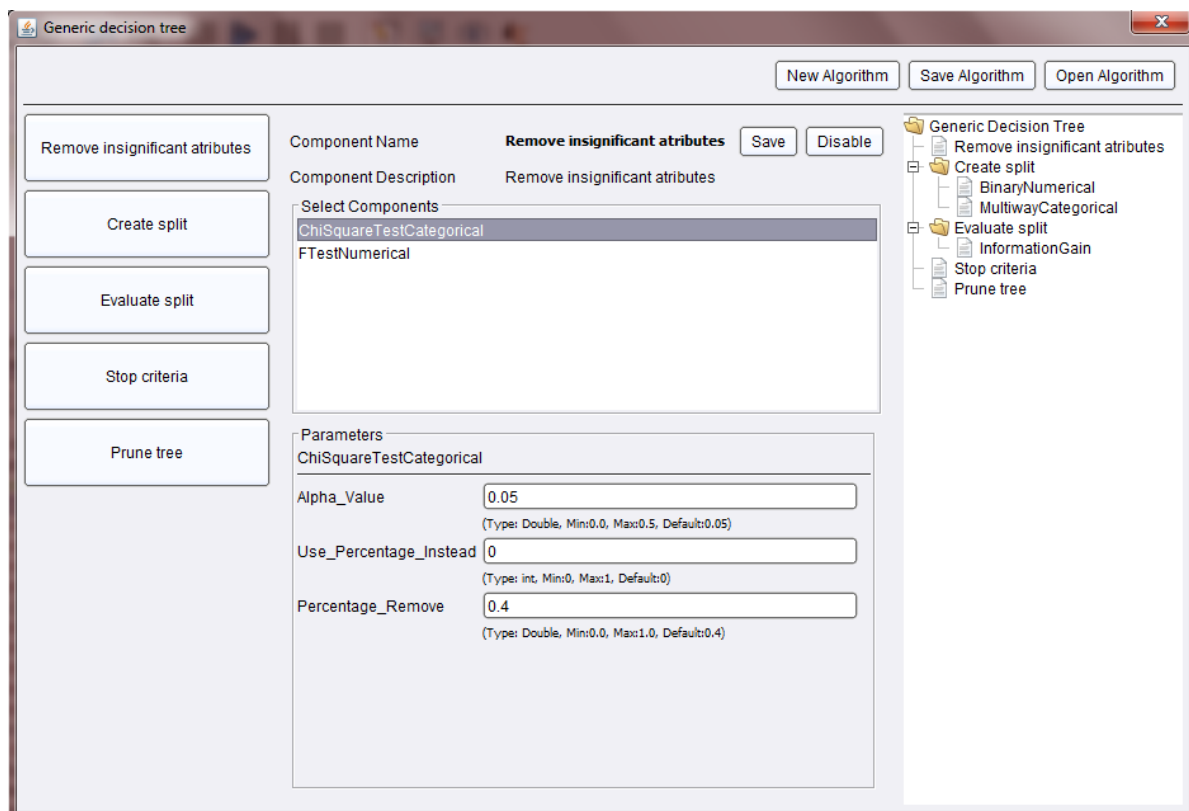


Figure 6 - WhiBo GDT user interface for design of decision tree algorithms

General procedure for designing new algorithms:

- Select sub-problem from left panel. When sub-problem is selected, possible solutions (RCs) are shown in central panel.
- Select RC (or components if multiple) for sub-problem from central panel. If parameters for component(s) are available, they will be shown in bottom part of central panel with their default values.
- Click on save component button. Components and defined parameters for selected sub-problem will be shown in the right panel as part of current GDT algorithm.
- This procedure should be repeated for every sub-problem (Create split and Evaluate split sub-problem are basic for decision tree growth and they must be defined. Definition of other sub-problems is optional). When all sub-problems, components and parameters are defined algorithm should be saved on file system (click on save button from upper panel). By default algorithms are saved with *.wba* (white box algorithm) extension.

## **WHIBO generic decision tree (GDT) evolutionary search operator GUI manual**

WHIBO generic decision tree (GDT) evolutionary search operator implements genetic algorithm which selects reusable components defined in *.ass* (algorithm search space) file.

Parameters:

- Algorithm search space file location – location of *.ass* file
- Parameters – list of parameters of genetic algorithm
- Wba file path macro name – macro pointing to *.wba* file
- Log file path – path where log file will be saved

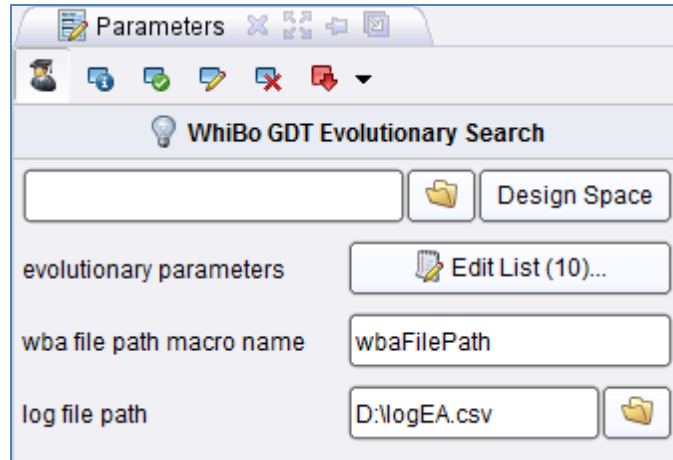


Figure 7 - Parameters panel for *WhiBo GDT Evolutionary Search*

Similarly like in WhiBo generic decision tree user interface contains four panels:

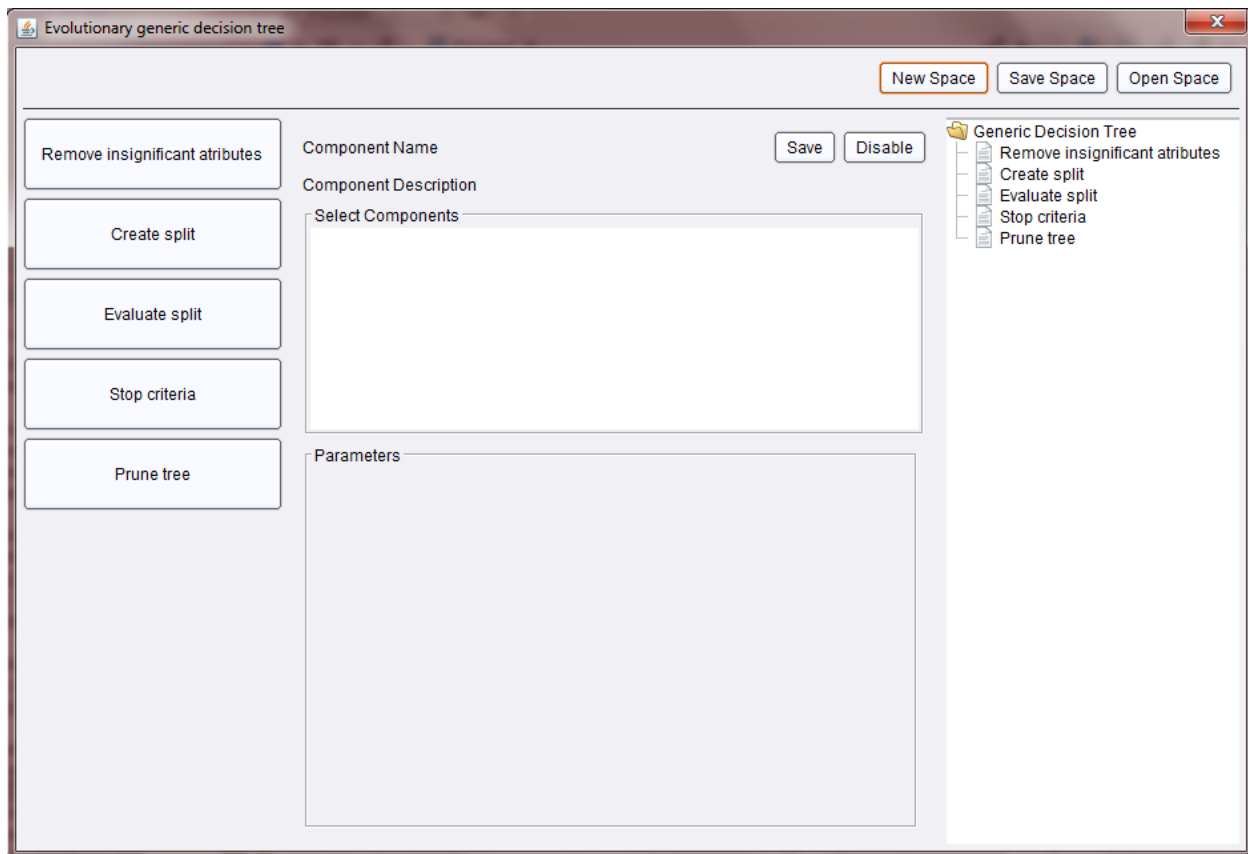
Left panel contains an array of buttons. Every button represents a concrete sub-problem for a decision-tree algorithm design.

Central panel contains:

- Available RCs of selected sub-problem from the left panel.
- Available parameters (if available) for selected RCs.
- Buttons for including or disabling a RC from the current decision tree structure.

Right panel shows current state of user designed algorithm (saved sub-problems, RCs and parameters).

Top panel contains options for creating new, saving current or opening existing generic decision tree algorithm.



**Figure 8 - WhiBo GDT evolutionary search user interface for design of algorithm search space**

General procedure for designing algorithm search space:

- Select sub-problem from left panel. When sub-problem is selected, possible solutions (RCs) are shown in central panel.
- Select RC (or components if multiple) for sub-problem from central panel. If parameters for component(s) are available, they will be shown in bottom part of central panel with lower and upper values selected. User can modify these values.
- Click on save component button. Components and defined parameters for selected sub-problem will be shown in the right panel as part of current GDT algorithm.
- This procedure should be repeated for every sub-problem (Create split and Evaluate split sub-problem are basic for decision tree growth and they must be defined. Definition of other sub-problems is optional). When all sub-

problems, components and parameters are defined algorithm should be saved on file system (click on save button from upper panel). By default algorithms are saved with *.ass* (algorithm search space) extension.

After definition of algorithm search space parameters for genetic algorithm should be defined.

Parameters:

- MAX\_ALLOWED\_EVOLUTIONS – maximal numbers of generations of genetic algorithms (default value - 50).
- POPULATION\_SIZE – number of units (decision trees) in one generation (default value - 30).
- MUTATION\_RATE – percentage of genes (components) will be changed (default value - 6).
- CROSSOVER\_RATE – rate of crossover of chromosomes in genetic algorithm (default value – 0.35)
- SWITCH\_FROM\_SURROGATE\_PERCENTAGE\_EVOLUTIONS – defines how many units should be removed from previous generation (default value – 0.4)
- SURROGATE\_PERCENTAGE – defines how many units should be selected from previous generation (default value – 0.4)
- mutateComponents – boolean value indicating weather reusable components should be mutated (default value - true).
- mutateParameters – boolean value indicating weather parameters should be mutated (default value - true).
- componentsMutationRate – mutation rate of components (default value - 1).
- parametersMutationRate – mutation rate of parameters (default value - 1).



| parameter name                              | values |
|---|--------|
| MAX_ALLOWED_EVOLUTIONS                      | 50     |
| POPULATION_SIZE                             | 30     |
| MUTATION_RATE                               | 6      |
| CROSSOVER_RATE                              | 0.35   |
| SWITCH_FROM_SURROGATE_PERCENTAGE_EVOLUTION: | 0.4    |
| SURROGATE_PERCENTAGE                        | 0.3    |
| mutateComponents                            | true   |
| mutateParameters                            | false  |
| componentMutationRate                       | 1      |
| parametersMutationRate                      | 1      |

Figure 9 - Parameters of genetic algorithm

## WHIBO testing environment manual

WhiBo provides operators for testing performance and significance of differences in algorithm performance.

*Custom cross validation with log* - implements cross validation with custom defined number of folds and number of iterations and also enables writing results in log in CSV format. The results are written in average, but also for every fold and iteration. This operator writes accuracy of classifier, but also: Maximum tree depth, weighted average tree depth, Total nodes, Total leaves, and Execution time.

Parameters:

- *Average\_performances\_only* – check if there is no need for logging the results for every fold and iteration.
- *Algorithm\_name* – name of the algorithm.
- *Dataset\_name* – name of the dataset.
- *Number\_of\_folds* – number of folds for cross-validation.
- *Number\_of\_repetitions* – number of repetitions for cross-validation.

- *Sampling\_type* – stratified sampling, linear sampling or shuffled sampling.
- *Log\_file\_details* – file path for logging detailed results.
- *Log\_file\_averages* - file path for logging average results.

The screenshot shows a 'Parameters' dialog box titled 'X-Validation with log'. It contains several settings:

- ☐ *keep example set*
- ☒ *average performances only*
- algorithm name: GDT
- dataset name: Iris
- number of folds: 2
- number of repetitions: 5
- sampling type: stratified sampling (dropdown menu)
- local random seed: 1
- log file details: desktop\LogDetails.csv (with a folder icon)
- log file averages: ktop\LogAverages.csv (with a folder icon)
- ☐ *parallelize training*
- ☐ *parallelize testing*

Figure 10 - Custom cross validation with log operator with parameters

*Significance 5X2 cross validation F-test* – This is the best significance tester for classifiers according to (Salzberg, 1999). The 5x2 cross validation F-test (Alpaydin E. (1999)) is testing significance of differences in algorithm performance.

Parameters:

- Alpha – significance parameter (Default value – 0.05).
- Local random seed – number used for initialization of pseudorandom number generator.
- *Sampling\_type* – stratified sampling, linear sampling or shuffled sampling.

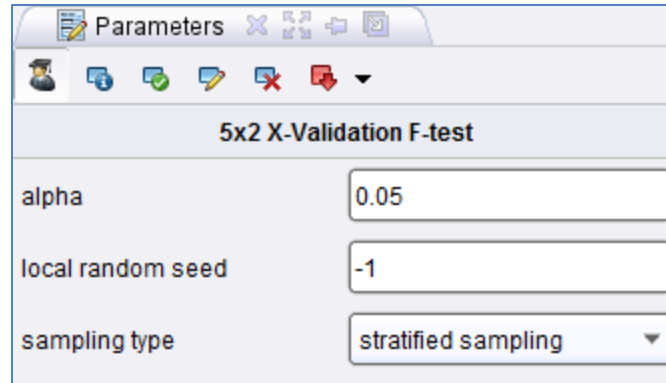


Figure 11 - Significance 5X2cv F-test operator with parameters

## Application examples

WhiBo GDT is implemented as RapidMiner operator. WhiBo decision tree operators require *ExampleSet* as input and produce *TreeModel* and *ExampleSet* on output, so they are compatible with all Rapid miner's evaluation and visualization operators.

For these examples we use "Iris" dataset from UCI repository as a data source (definition of data source can be done through RapidMiner's sample data repository).

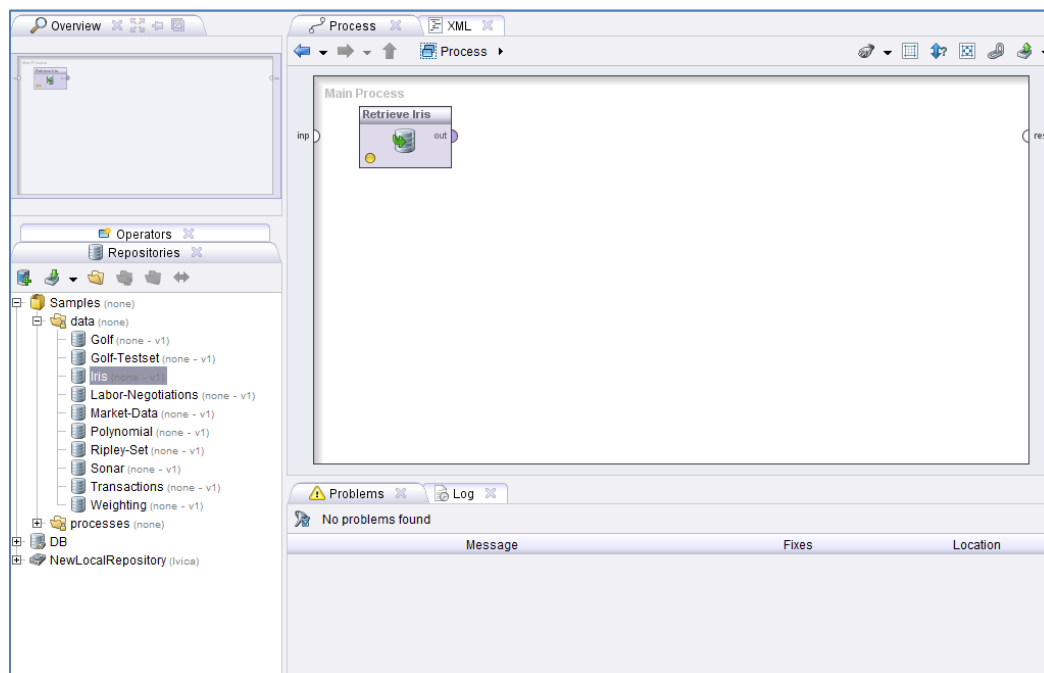


Figure 12 - Basic definition of RapidMiner process

On the lower left side of the screen local repository can be seen. From there, "Iris" dataset was dragged to Main Process panel. With that step input *ExampleSet* is defined.

## White-box component based design and application

Using WhiBo GDT with RapidMiner will be explained on examples of creating well-known algorithms, modifying these algorithms and designing new algorithms.

When *ExampleSet* is defined, add GDT operator to root process. GDT can be found in *WhiBo/GDT Operators* operator group.



Figure 13 - Adding GDT operator into stream

When the example source is defined and Generic Tree operator is added in process, new generic decision tree can be designed, by clicking on Design new algorithm button.

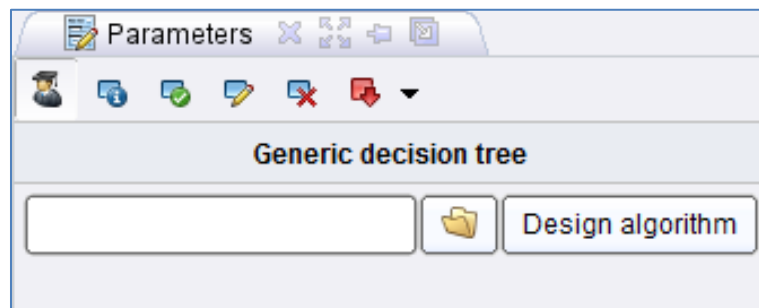


Figure 14 - Parameters panel for GDT operator

## Recreation of well-known algorithms with component based approach

Application of white-box approach will be first explained on recreation of well-known algorithms.

### CART algorithm

First, define Create split sub-problem:

- Click on Create split sub-problem on the left panel.
- Select *BinaryNumerical* and *BinaryCategorical* components from central panel (multiple components for one sub-problem are selected by holding CTRL key and clicking on components).
- Click on save component button from central panel.

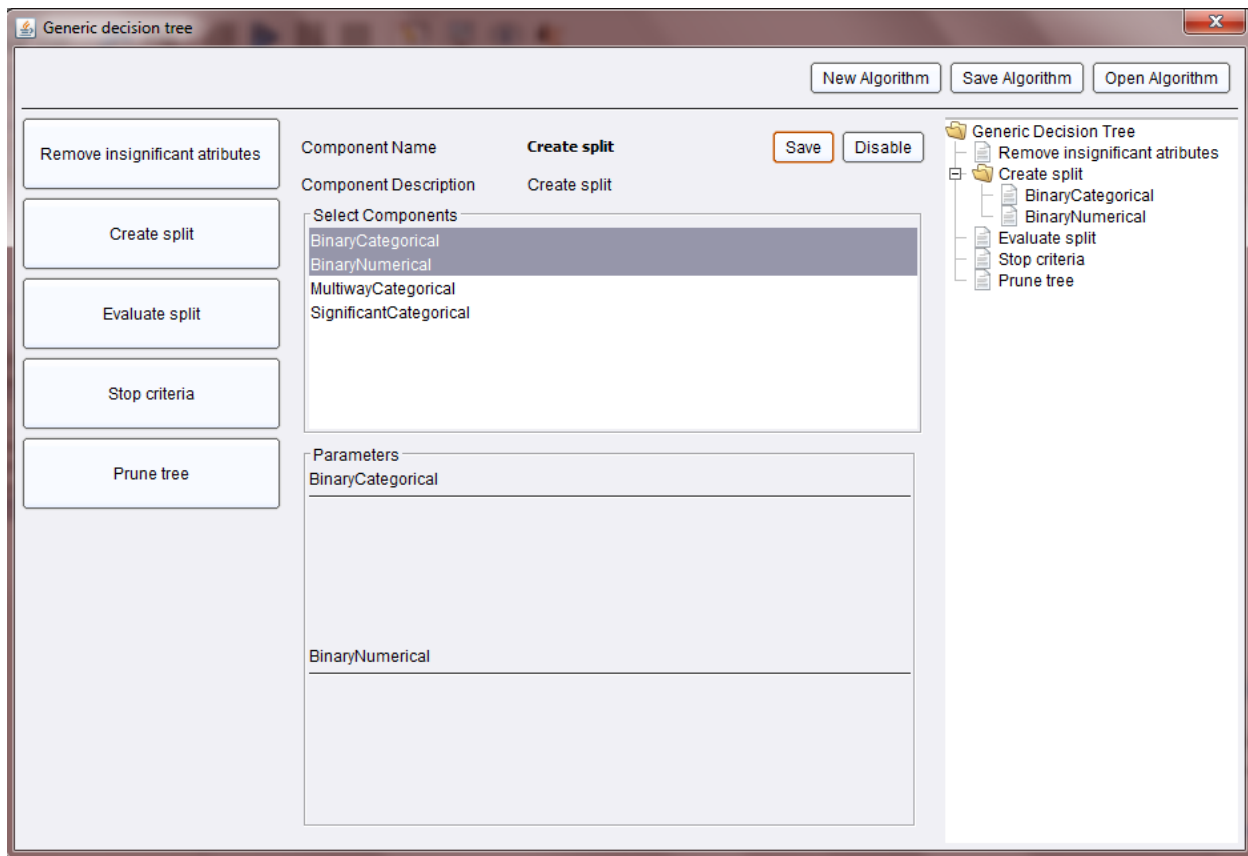


Figure 15 - Definition of *Create split* sub-problem for CART algorithm

On the right panel defined components for a sub-problem is visualized through a Tree view (Figure above).

Next step is definition of evaluate split sub-problem.

- Click on *Evaluate split* sub-problem.
- Select *Gini index* component.
- Click on save component button.

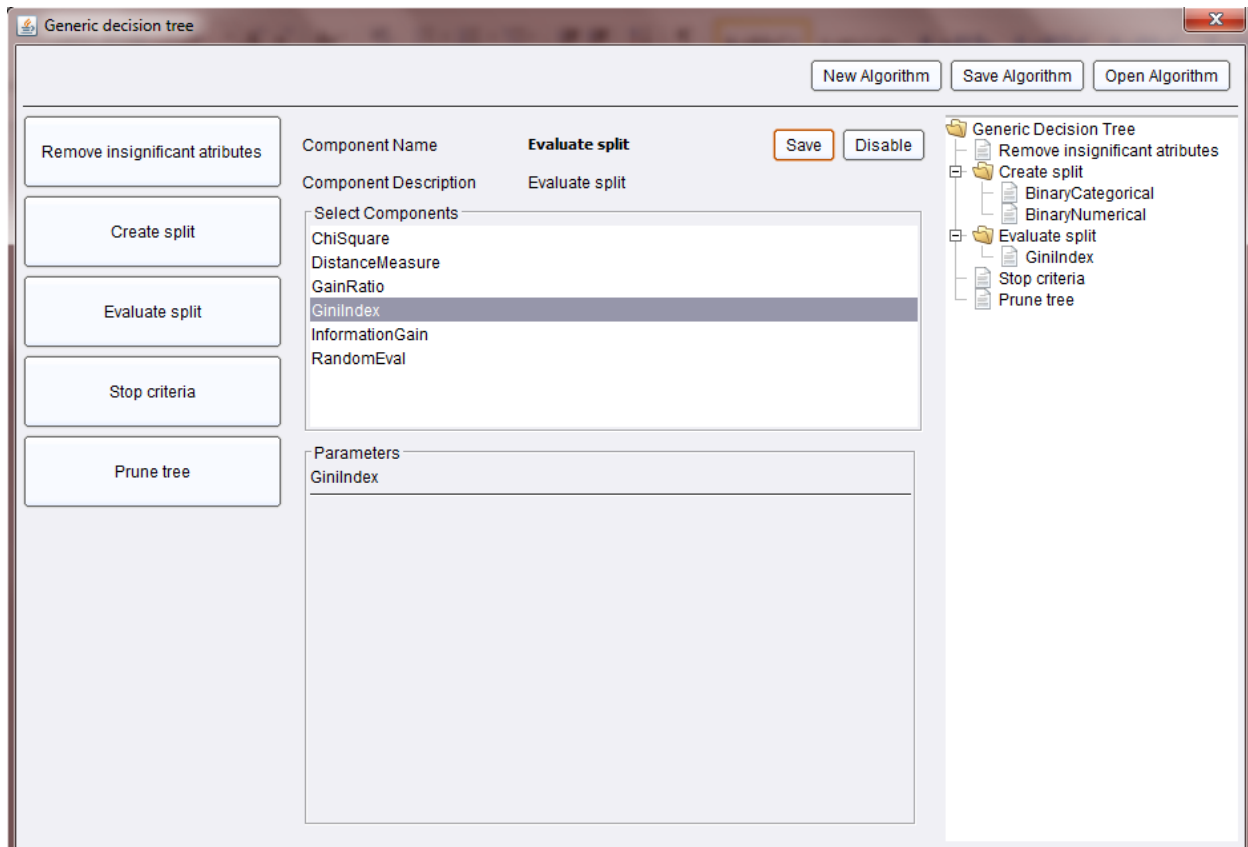


Figure 16 - Definition of *Evaluate split* sub-problem for CART algorithm

Now, the basic components for CART algorithm are defined. Before saving the algorithm we will define Stop criteria sub-problem:

- Click on *Stop criteria* sub-problem.
- Select *Tree depth* component
- Set *Tree\_Depth* parameter on 5 (default value is 10).
- Click on save component button.

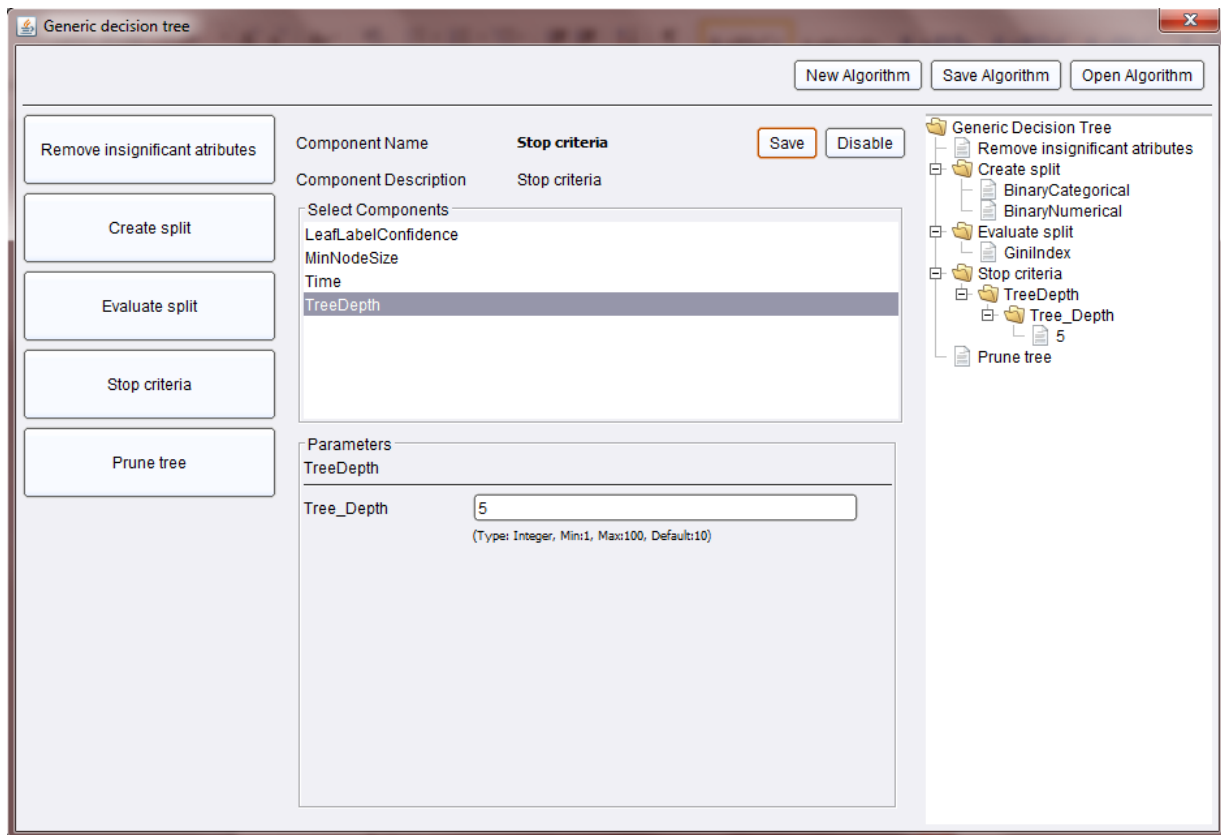


Figure 17 - Definition of *Stop criteria* sub-problem for CART algorithm

Finally Cart algorithm with tree depth stopping criteria is defined and can be saved on file system.

Click on Save algorithm button from upper panel.

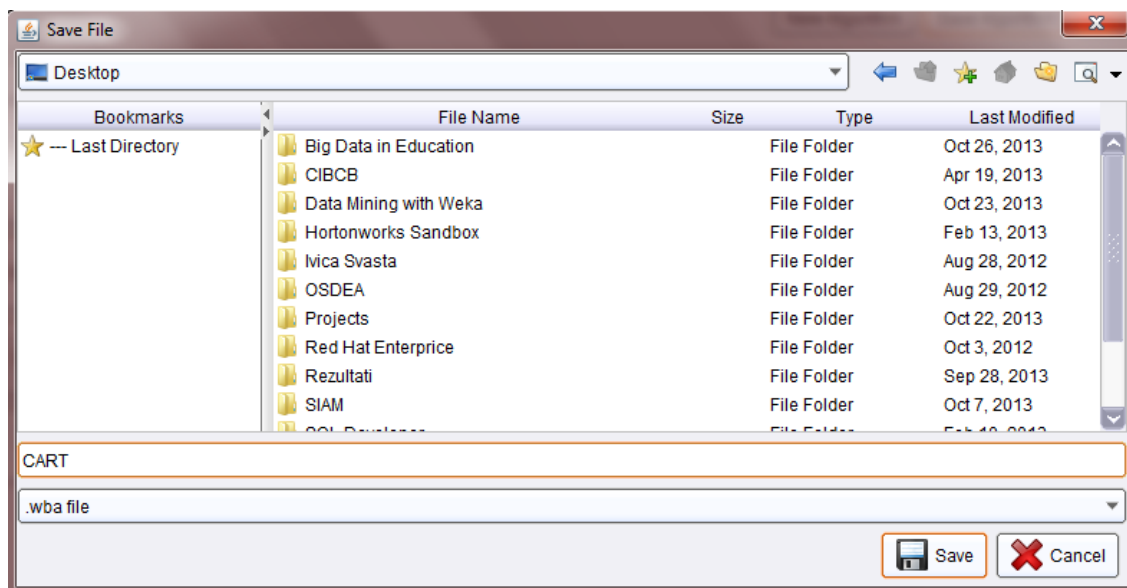


Figure 18 - Saving CART algorithm on file system

After saving algorithm it must be loaded in GDT GUI clicking on folder button in parameters panel.

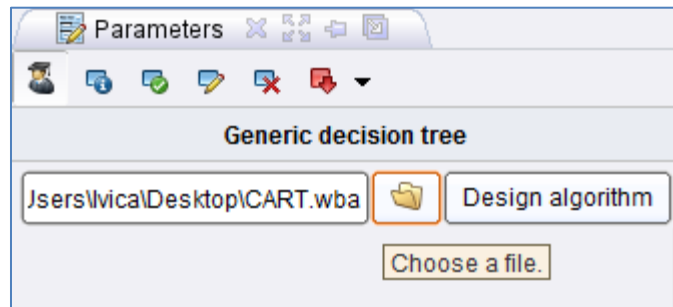


Figure 19 - Loading CART algorithm in GDT operator

When stream is executed, graphic a text tree model will be shown.

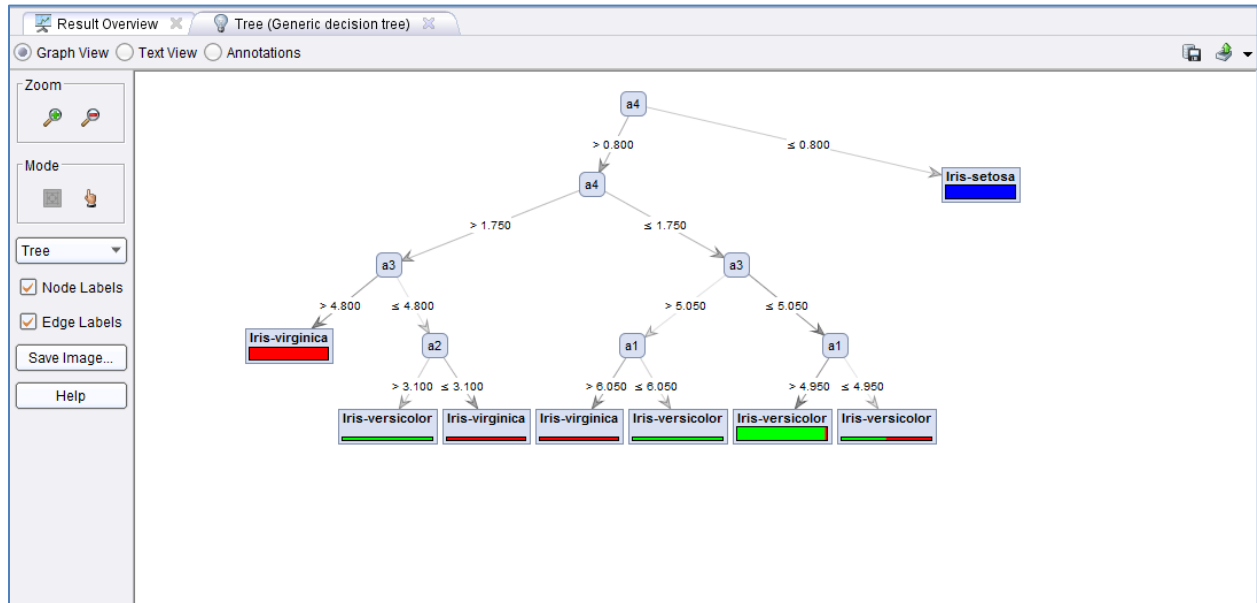


Figure 20 - Result of executed CART algorithm on Iris dataset



## C4.5 algorithm

First, define Create split sub-problem:

- Click on Create split sub-problem on the left panel.
- Select *BinaryNumerical* and *MultiwayCategorical* components from central panel (multiple components for one sub-problem are selected by holding CTRL key and clicking on components)
- Click on save component button from central panel

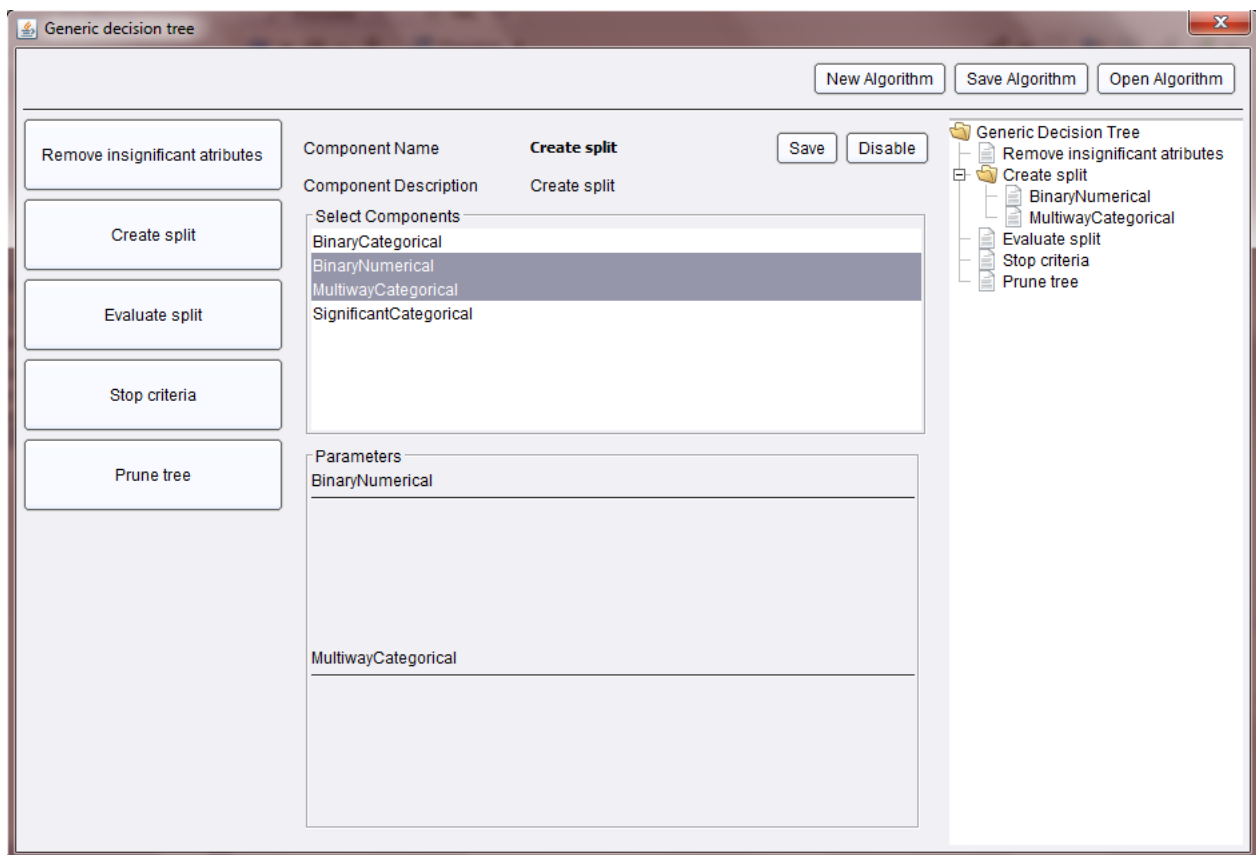


Figure 21 - Definition of *Create split* sub-problem for C4.5 algorithm

Second, define Create split sub-problem:

- Click on *Evaluate split* sub-problem on the left panel.
- Select *GainRatio* component from central panel.
- Click on save component button from central panel.

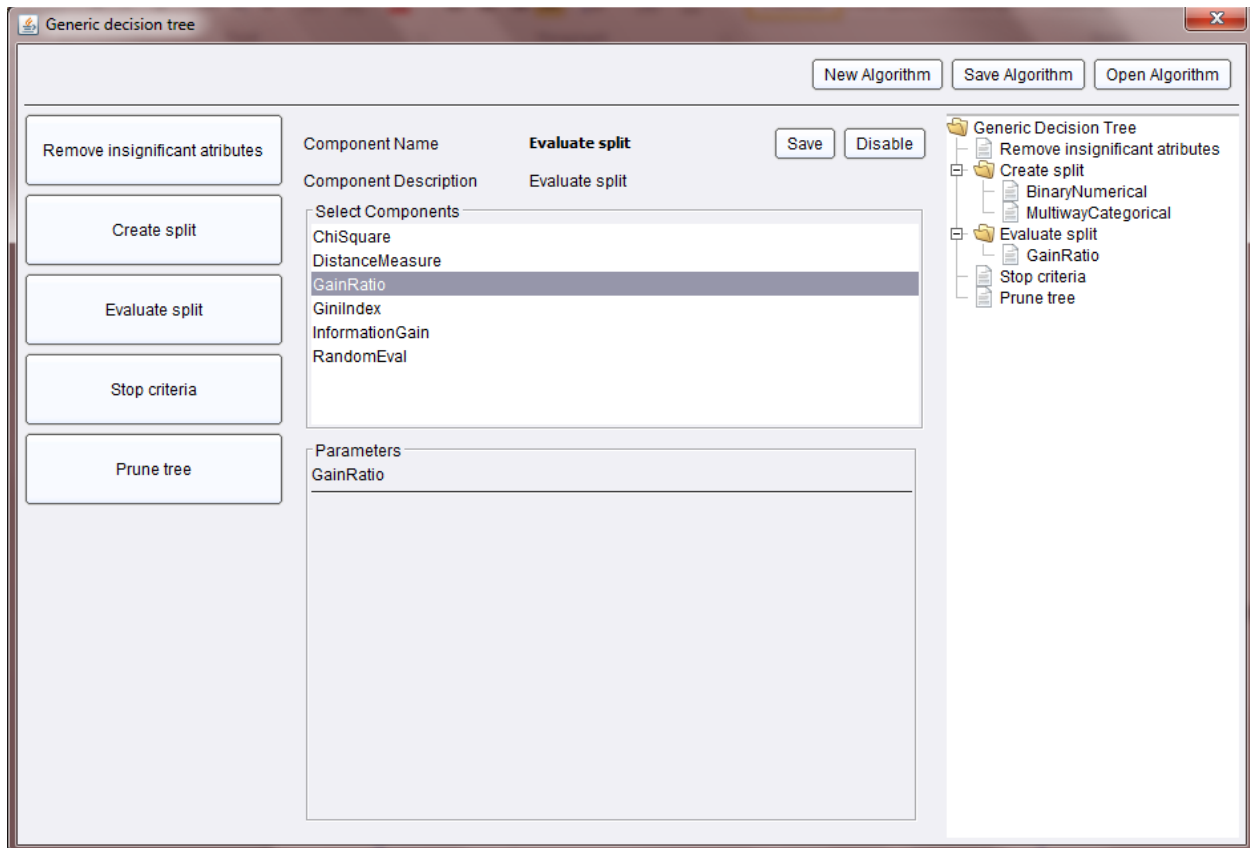


Figure 22 - Definition of *Evaluate split* sub-problem for C4.5 algorithm

Third, define *Stop criteria* sub-problem:

- Click on *Stop criteria* sub-problem on the left panel.
- Select *TreeDepth* component from central panel.
- Set *Tree\_Depth* parameter for example on 10.
- Click on save component button from central panel.

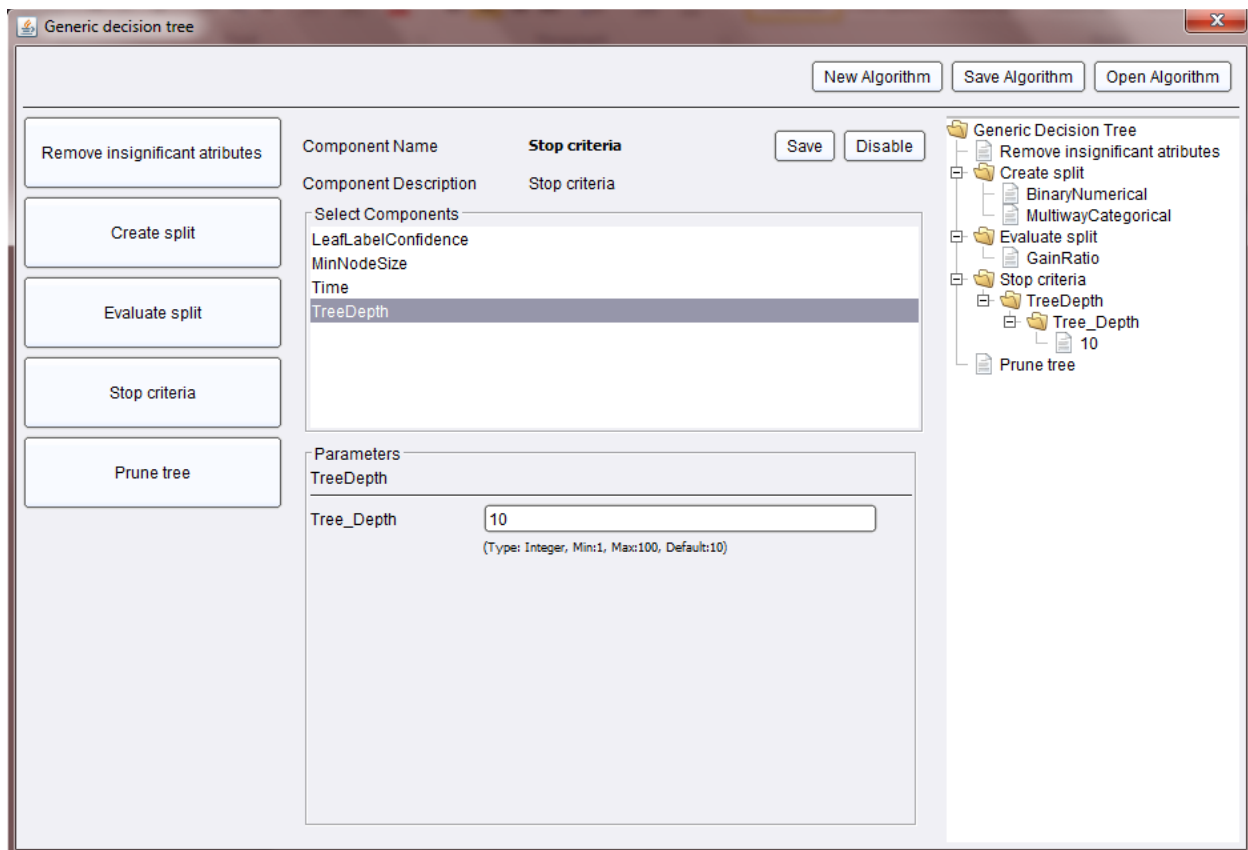


Figure 23 - Definition of *Stop criteria* sub-problem for C4.5 algorithm

Fourth, define *Prune tree* sub-problem:

- Click on *Prune tree* sub-problem on the left panel.
- Select *PessimisticError* component from central panel.
- Set *Confidence\_Level* parameter on 0.2.
- Click on save component button from central panel.

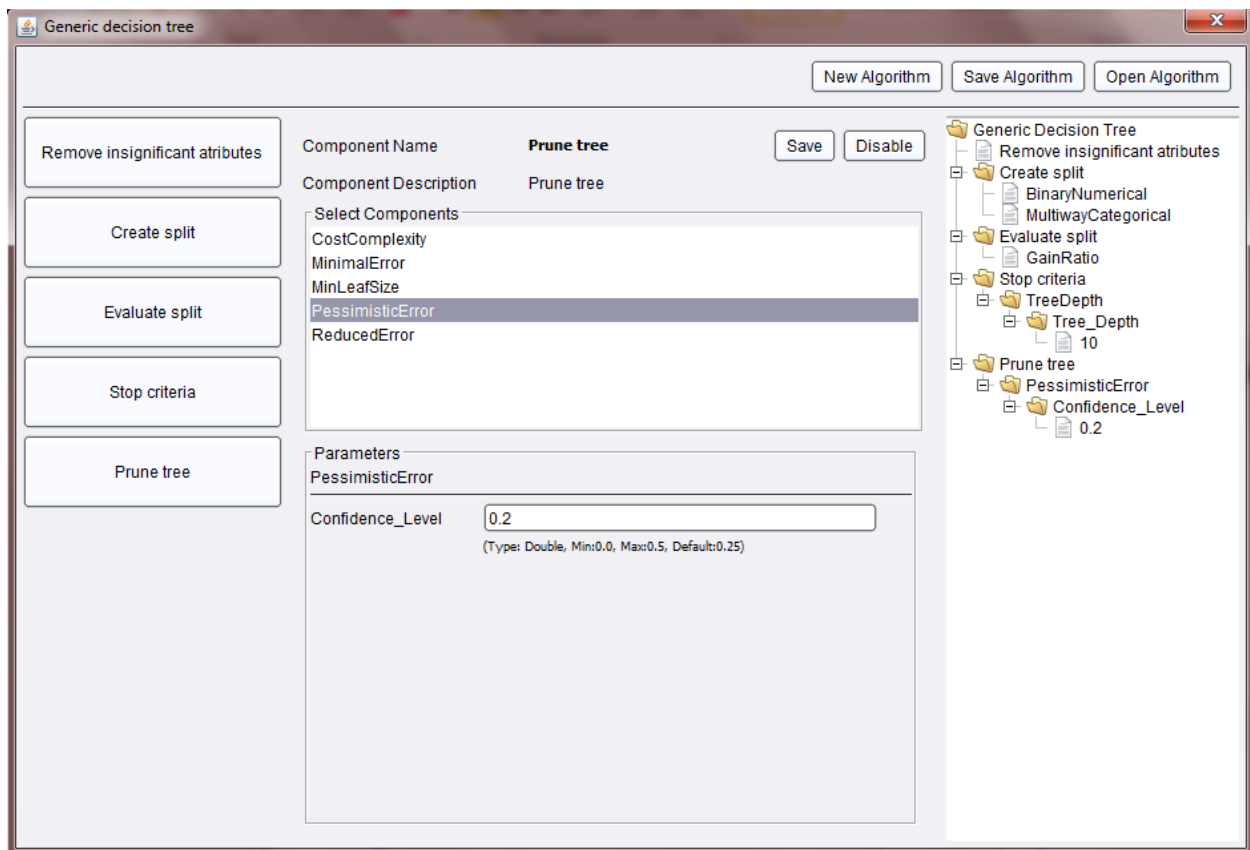


Figure 24 - Definition of *Prune tree* sub-problem for C4.5 algorithm

Finally, C4.5 algorithm is defined and can be saved on file system. Click on Save algorithm button from upper panel. When algorithm is defined, load it from file system and execute stream.

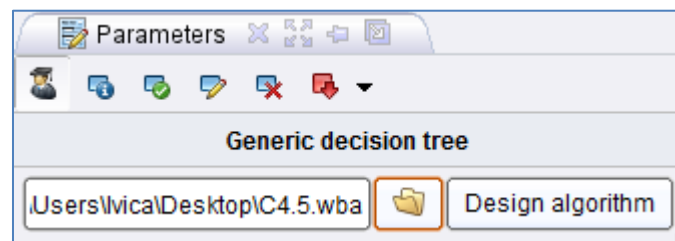


Figure 25 - Loading C4.5 algorithm in GDT operator

When stream is executed, graphic a text tree model will be shown.

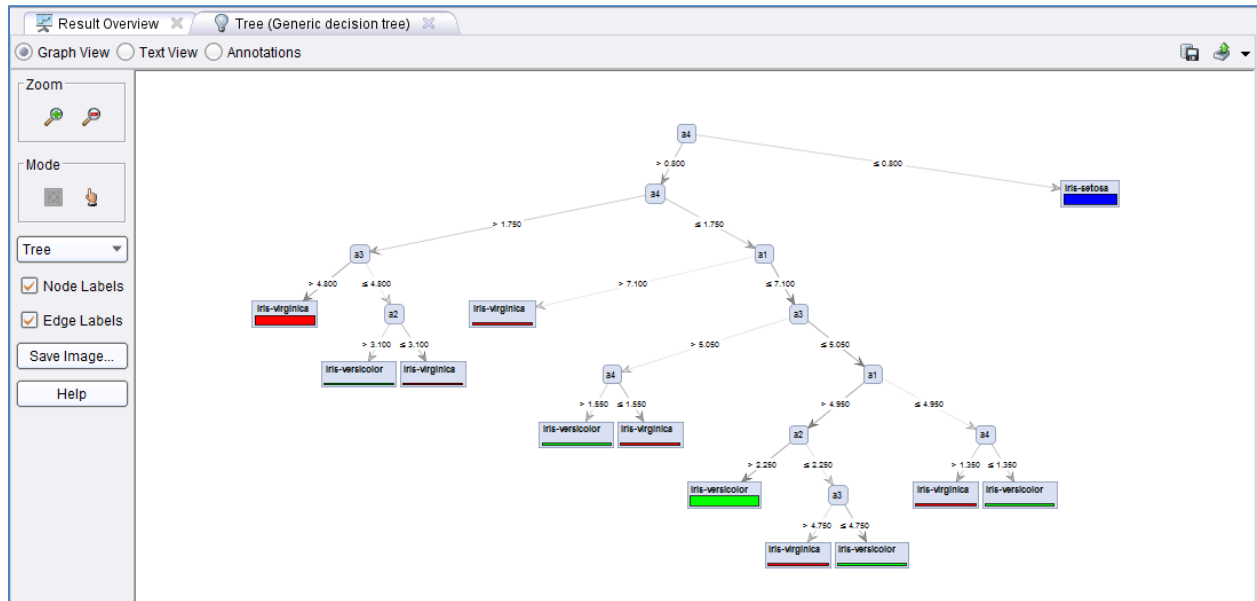


Figure 26 - Result of executed C4.5 algorithm on Iris dataset

## CHAID algorithm

First, define Create split sub-problem:

- Click on Create split sub-problem on the left panel.
- Select *BinaryNumerical* and *SignificantCategorical* components from central panel (multiple components for one sub-problem are selected by holding CTRL key and clicking on components)
- Set default parameters *Merge\_Alpha\_Value* and *Split\_Alpha\_Value* for *SignificantCategoricalComponent*.
- Click on Save component button from central panel.

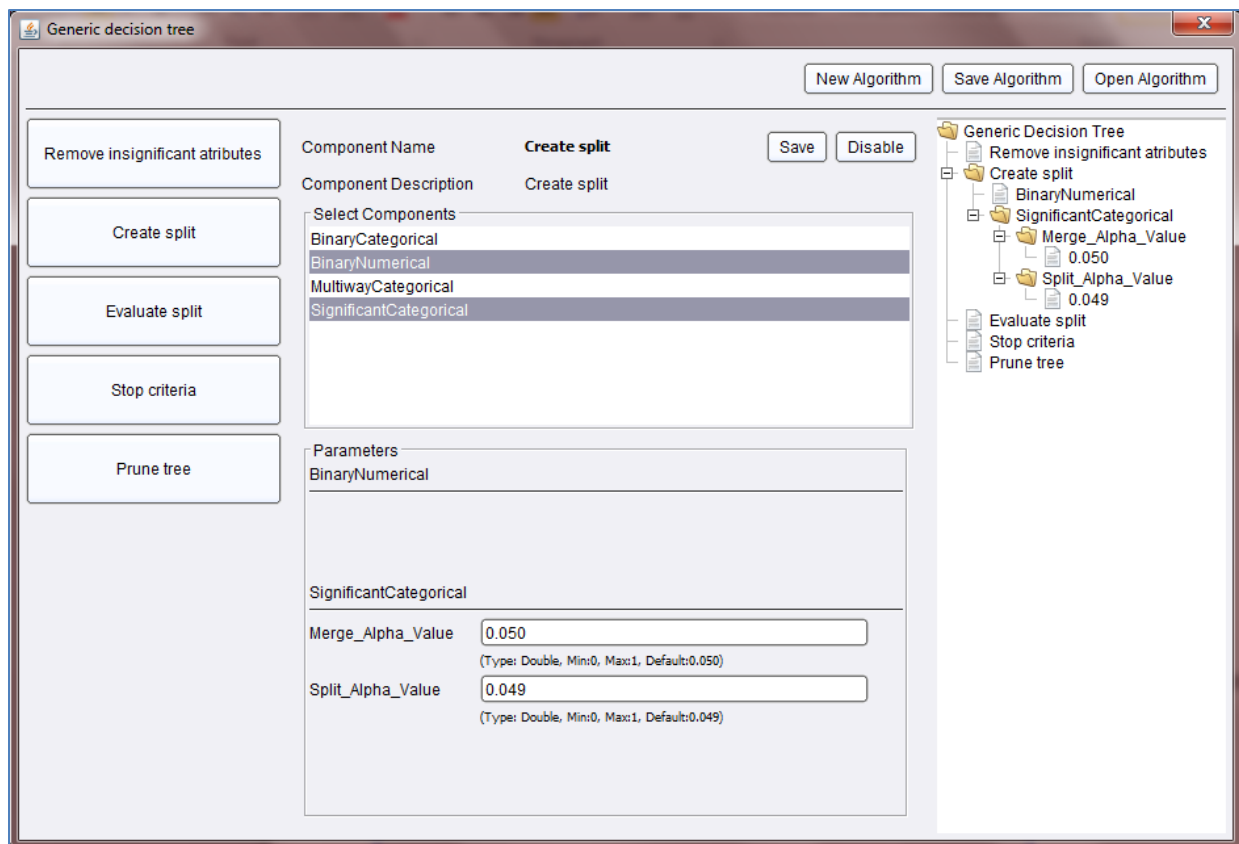


Figure 27 - Definition of *Create split* sub-problem for CHAID algorithm

Second, define Evaluate split sub-problem:

- Click on *Evaluate split* sub-problem on the left panel.
- Select *ChiSquare* component from central panel.
- Click on save component button from central panel.

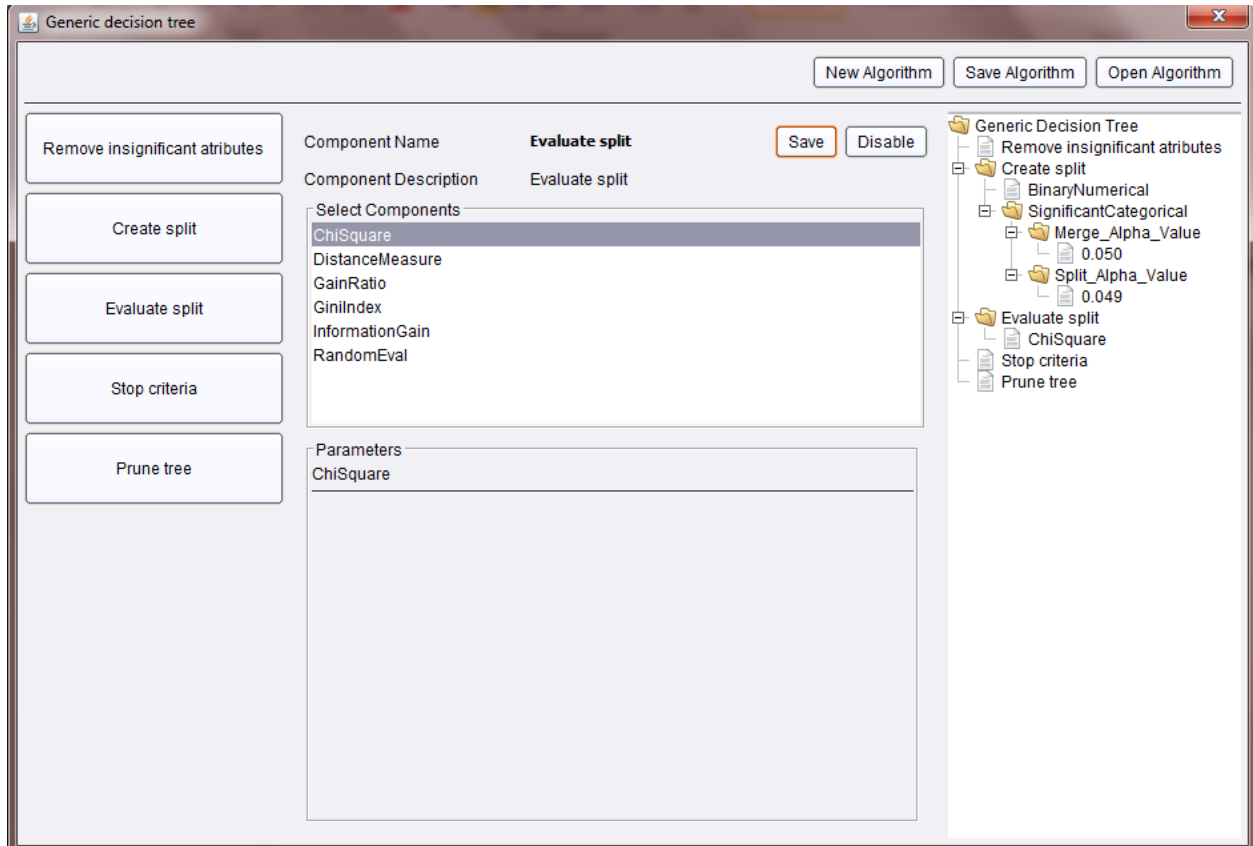


Figure 28 - Definition of *Evaluate split* sub-problem for CHAID algorithm

Third, define *Stop criteria* sub-problem:

- Click on *Stop criteria* sub-problem on the left panel.
- Select *TreeDepth* component from central panel.
- Set *Tree\_Depth* parameter for example on 5.
- Click on save component button from central panel.

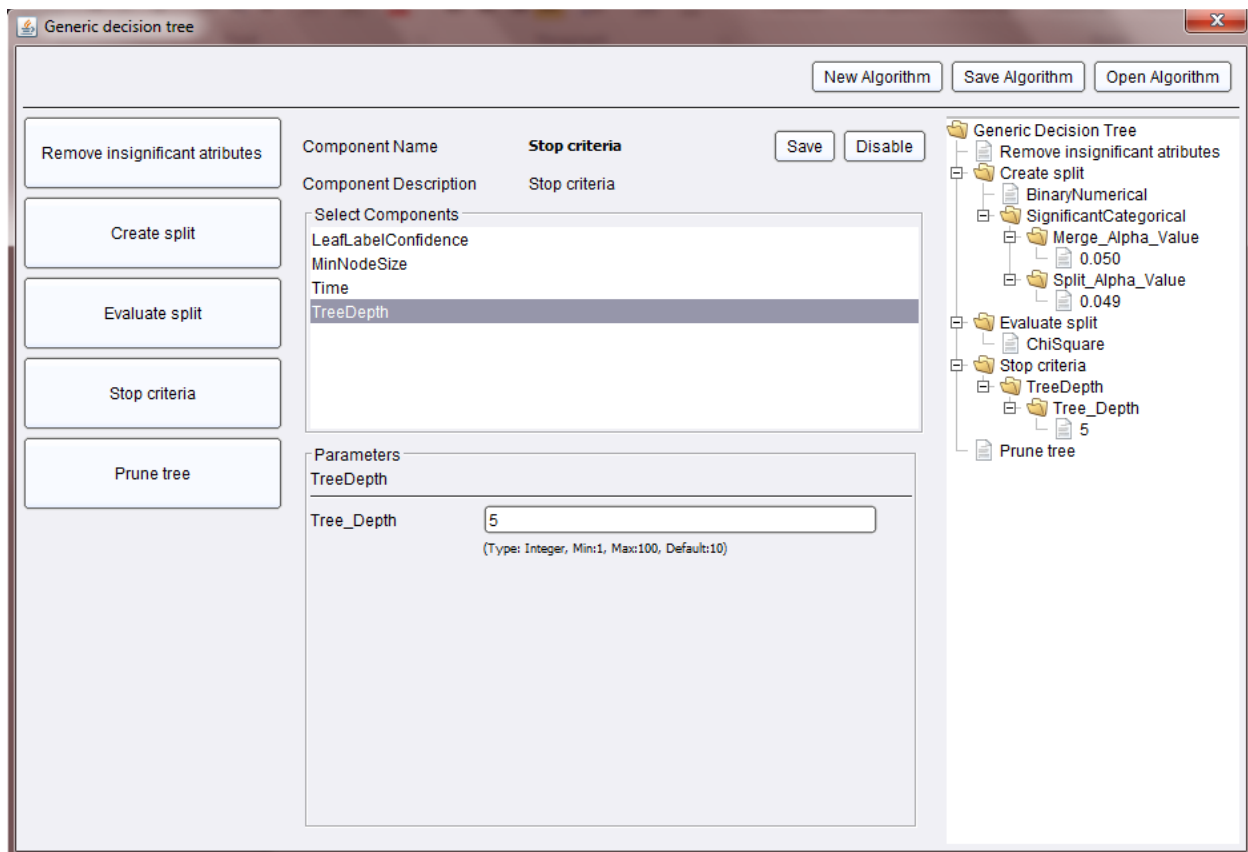


Figure 29 - Definition of *Stop criteria* sub-problem for CHAID algorithm



Fourth, define *Prune tree* sub-problem:

- Click on *Prune tree* sub-problem on the left panel.
- Select *PessimisticError* component from central panel.
- Set *Confidence\_Level* parameter for example on 0.2.
- Click on save component button from central panel.

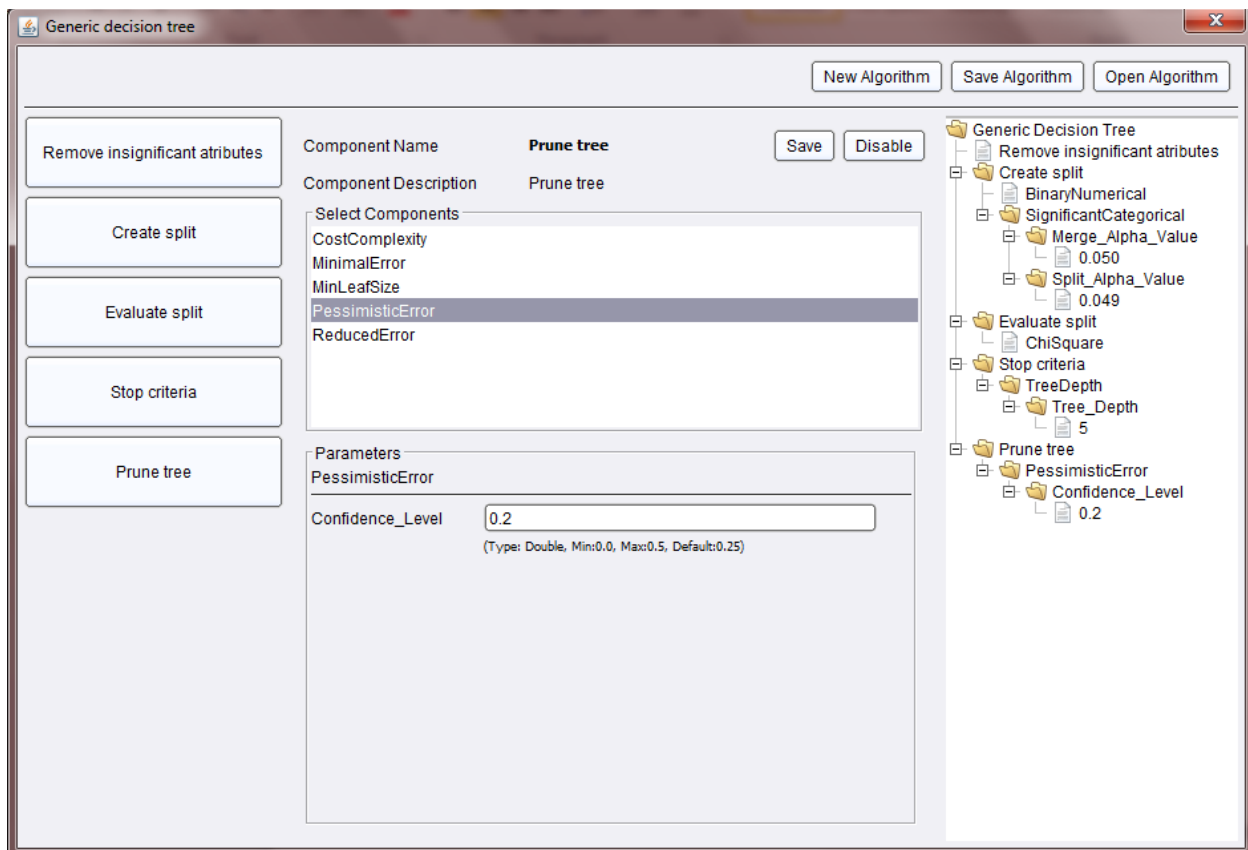


Figure 30 - Definition of *Prune tree* sub-problem for CHAID algorithm

Finally, CHAID algorithm is defined and can be saved on file system. Click on Save algorithm button from upper panel. When algorithm is defined, load it from file system and execute stream.

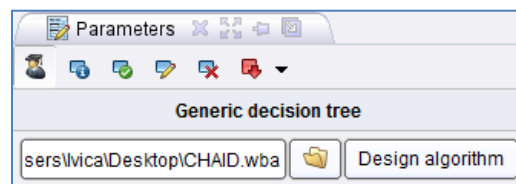


Figure 31 - Loading CHAID algorithm in GDT operator

When stream is executed, graphic a text tree model will be shown.

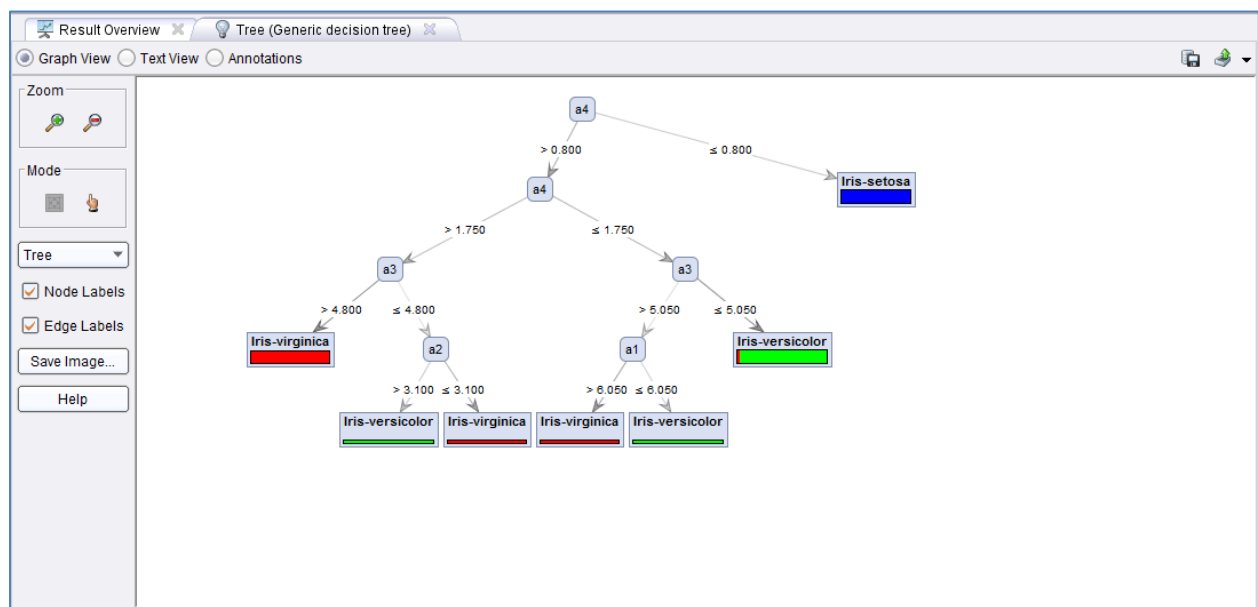


Figure 32 - Result of executed CHAID algorithm on Iris dataset

## Modifying generic decision tree algorithms

Algorithms created through WhiBo interface could be easily modified by parameters, sub-problems or components. WhiBo algorithms are saved on a file system by .wba (WhiBo algorithm) extension. Existing algorithms can be loaded for editing by clicking on *Open algorithm* button from the top panel of WhiBo interface.

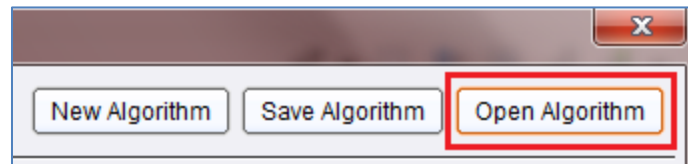


Figure 33 - Opening existing algorithm for modification

In this section it will be explained modifying of CHAID algorithm that was created and saved in previous example.

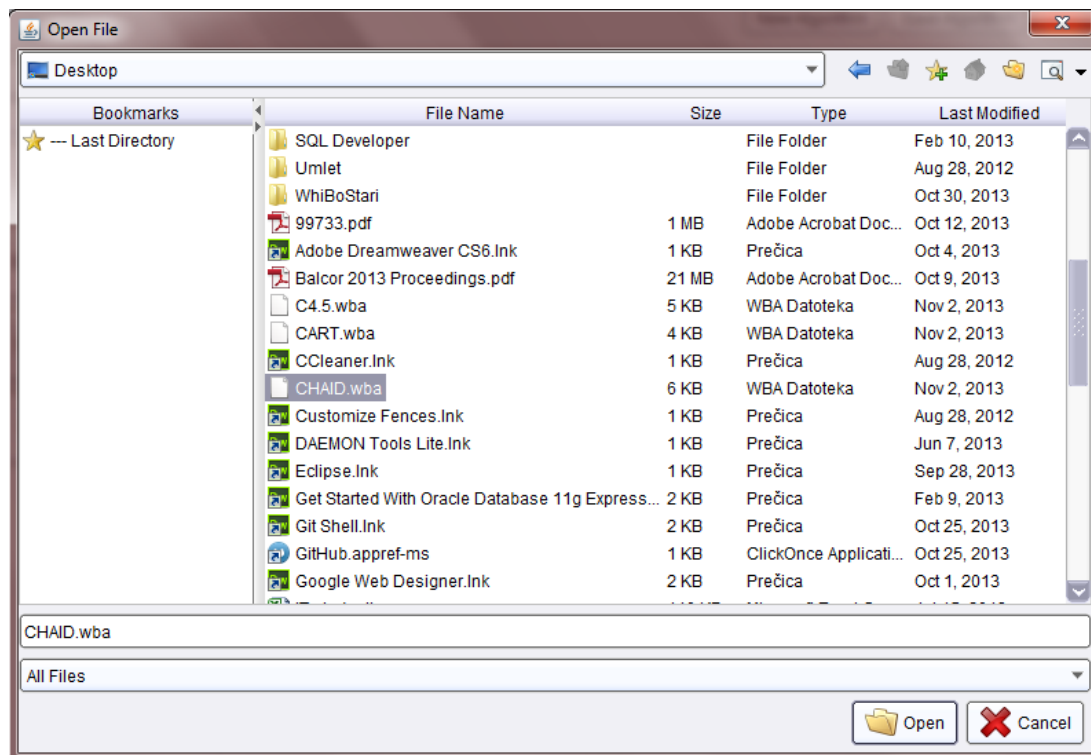


Figure 34 - Selecting existing algorithm from file system

When existing algorithm is opened, its sub-problems and RCs are shown in a tree view on the right panel of WhiBo interface.

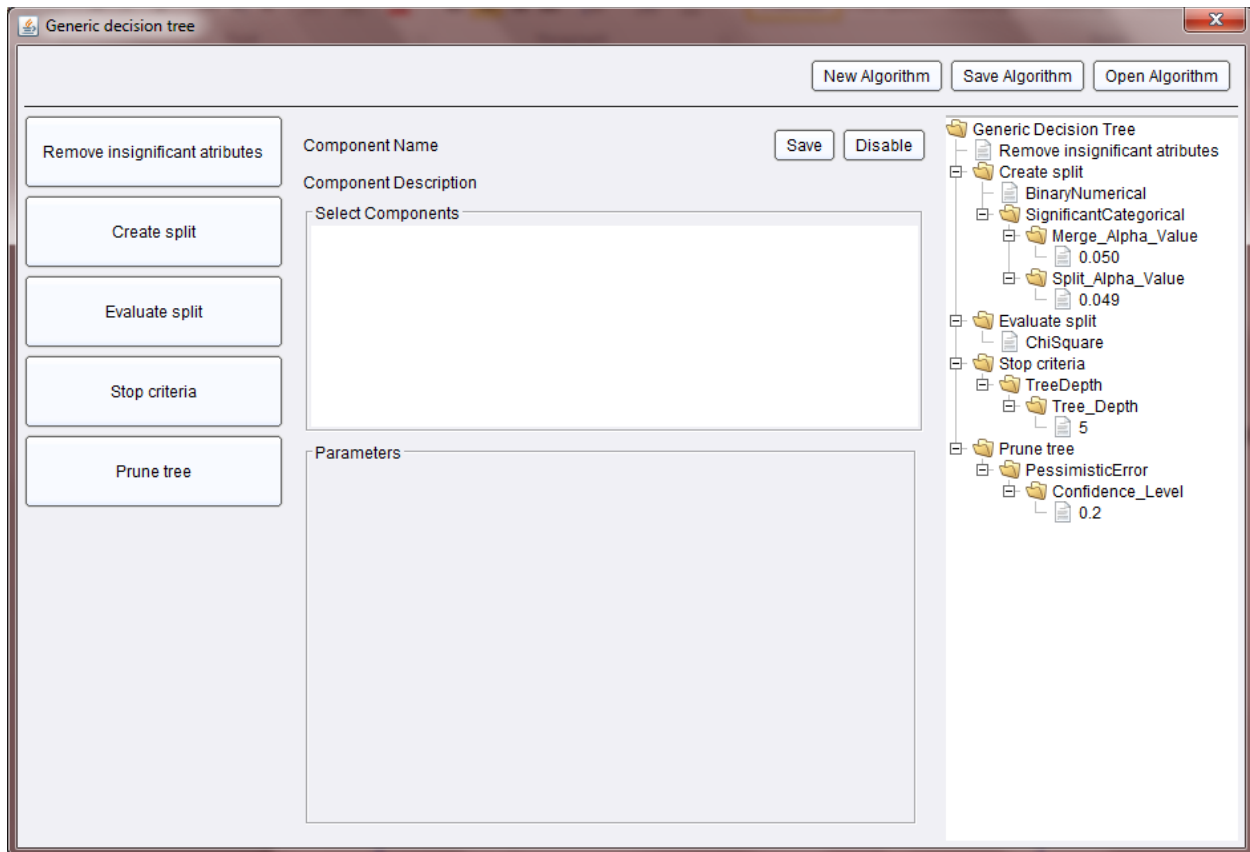


Figure 35 - Loaded algorithm in GDT GUI

Modifying of existing algorithms is done the same way as creating of new algorithms. In this example we will show how to

- modify parameter of already defined component,
- change component of defined sub-problem,
- adding new sub-problem and component for its solution.

## Modifying parameters

In this example we will modify *Merge\_Alpha\_Value* and *Split\_Alpha\_Value* parameters of *Create split – SignificantCategorical* component and *Tree\_Depth* parameter of *Stop criteria - TreeDepth*:

- Click on *Create split* sub-problem on the left panel.
- Set *Merge\_Alpha\_Value* on 0.03.
- Set *Split\_Alpha\_Value* on 0.02.
- Click on save component button from central panel.
- Click on *StopCriteria* sub-problem on the left panel.
- Set *Tree\_Depth* parameter on 4.
- Save algorithm as *CHAIDModifiedParameters* by clicking on *Save algorithm* button from top panel.

Result from executed algorithm is shown on figure below.

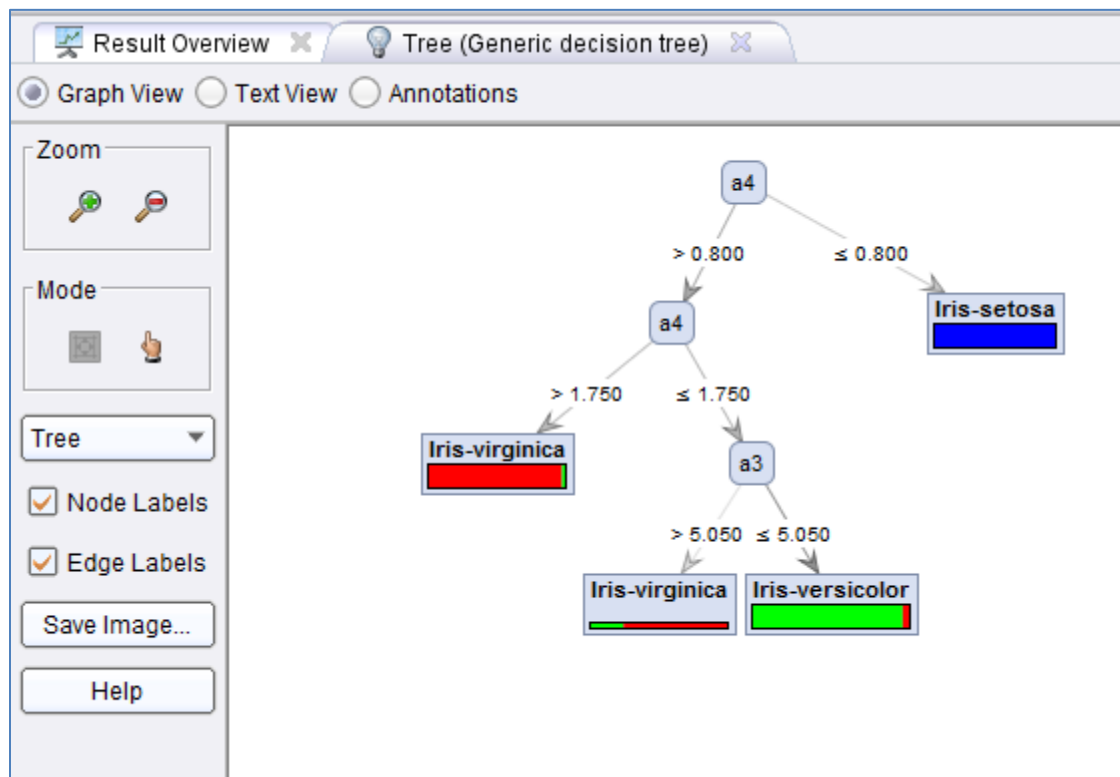


Figure 36 - Loaded algorithm in GDT GUI

## Replacing components for sub-problem

In this example we will replace Evaluation measure component in CHAIDModifiedParameters algorithm that is created in previous subsection.

- Click on *Evaluate split* sub-problem on the left panel.
- Select DistanceMeasure component from central panel.
- Click on save component button from central panel.
- Save algorithm as *CHAIDModifiedParametersDistance* by clicking on *Save algorithm* button from top panel.

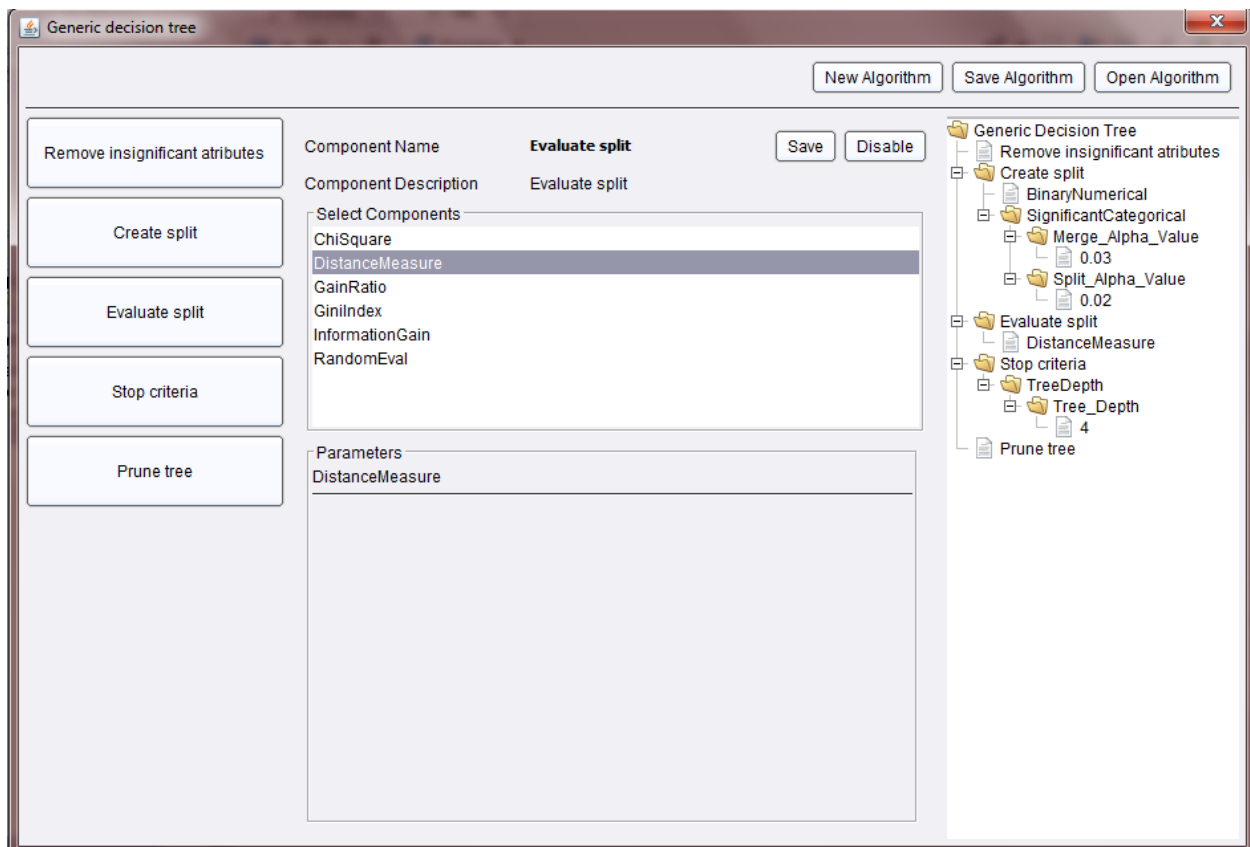


Figure 37 - Replacing components in CHAID algorithm in GDT GUI

Result from executed algorithm is shown on figure below.

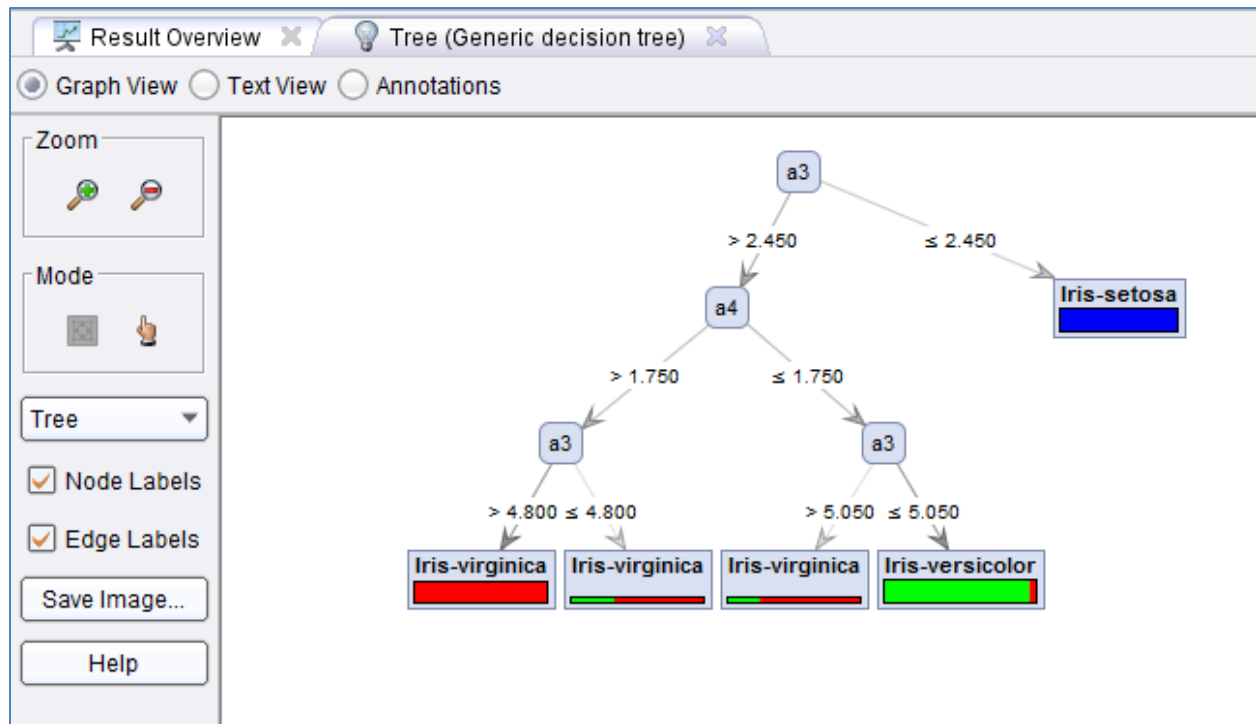


Figure 38 - Result of modified CHAID algorithm

## Adding new sub-problem and component to an existing algorithm

Besides changing of parameters and components existing algorithms could be extended with new sub-problems and components. We will explain this extension by adding Prune tree sub-problem to CHAID algorithm that is defined in previous subsection.

- Load CHAID algorithm from file system by clicking on *Open algorithm* button from top panel.
- Select *Prune tree* sub-problem from left panel.
- Click disable.
- Select *MinLeafSize* component from central panel.
- Set *Size\_Of\_Leaf* parameter to 15.
- Click on save component button from central panel.
- Save algorithm as *CHAIDPrune* by clicking on *Save algorithm* button from top panel.

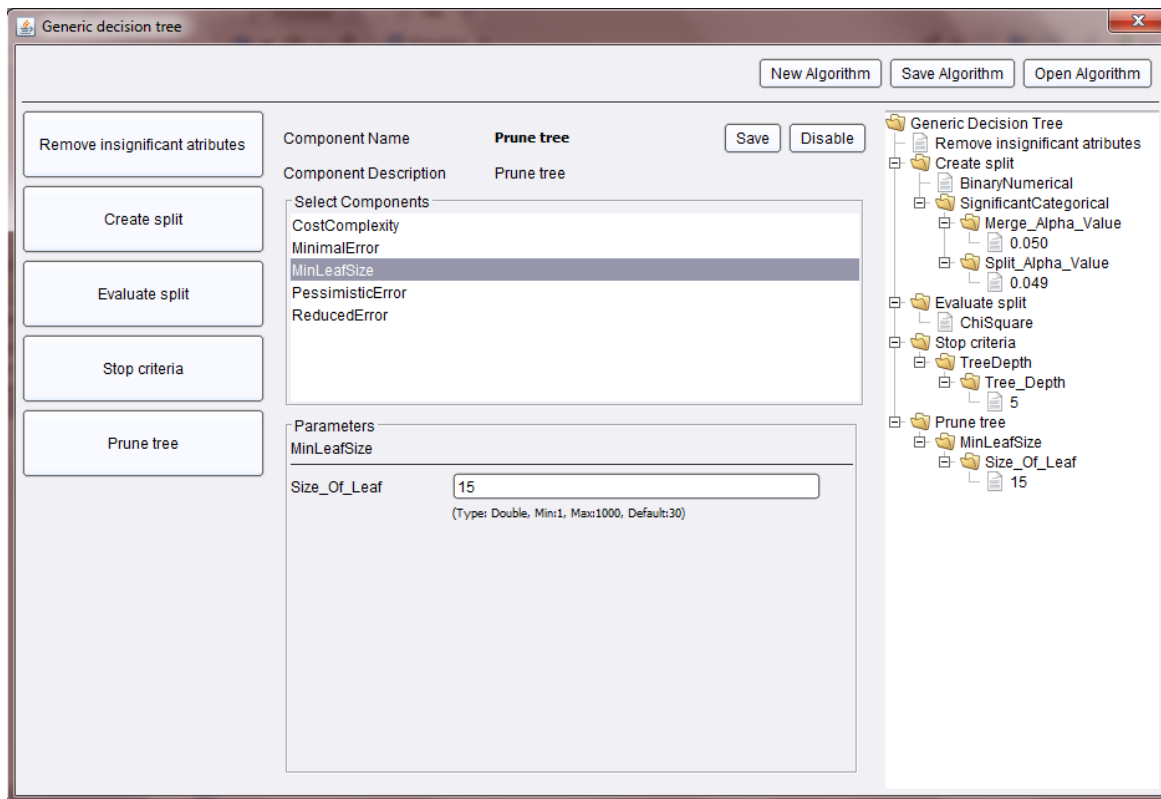


Figure 39 - Adding new sub-problem in existing algorithm

Result from executed algorithm is shown on figure below.

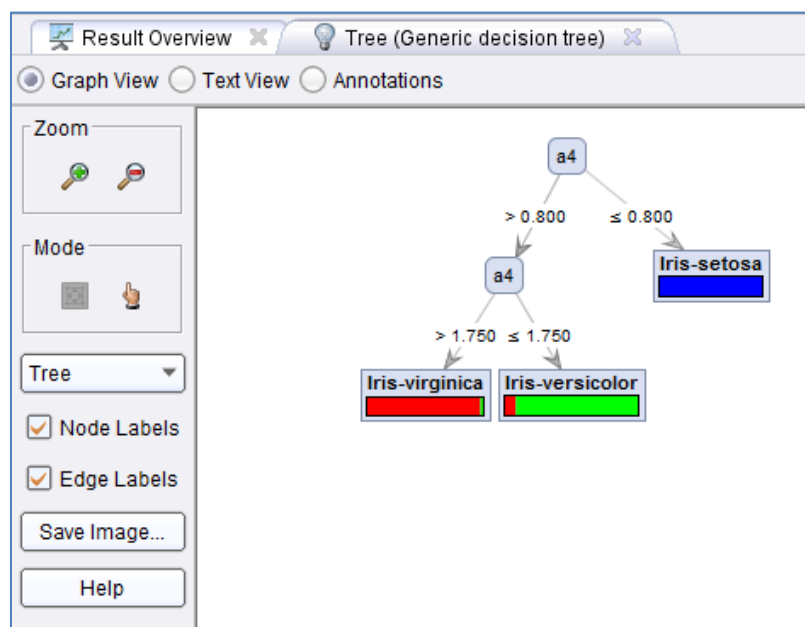


Figure 40 - Result of executed CHAID with prune algorithm on Iris dataset



Besides recreation and modification of existing algorithms WhiBo also enables:

- Design of new algorithms, by combination of components that are derived from well-known algorithms (C4.5, CART, CHAID) or partial algorithm improvements (e.g. distance measure).
- Incorporating partial improvements of algorithms that can be found in literature, but are not incorporated in any specific algorithm (e.g. Distance evaluation measure).
- Incorporating a new sub-problem in an algorithm (e.g. Remove insignificant attributes)
- Multiple component selection for sub-problem - it is possible to define more Splitting components stopping criteria.

## Generic decision tree evolutionary search design and application

WhiBo evolutionary search GDT is implemented as RapidMiner operator chain. WhiBo evolutionary search decision tree operators require *ExampleSet* as input and produce *TreeModel*, *ExampleSet* and *Performance* on output.

For these examples we use “Weighting” dataset from RapidMiner’s sample data repository.

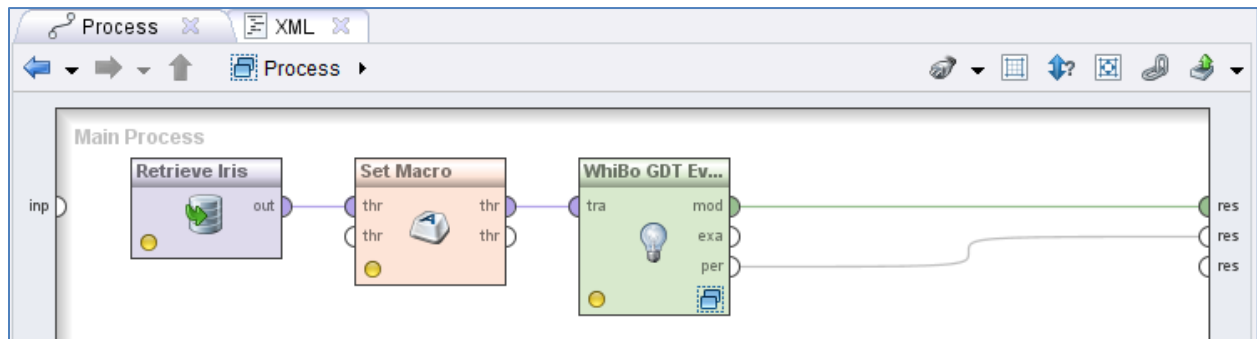


Figure 41 - Main process for WhiBo evolutionary search decision tree

Main process should contain at least three operators. Those are dataset (in this case Weighting dataset), Set Macro and WhiBo GDT Evolutionary Search. Since WhiBo GDT Evolutionary Search is operator chain it contains subprocess. Inside of this subprocess Generic decision tree should be placed.

After loading dataset Macro must be provided. Macro points to WhiBo algorithm file which is needed to *GDT Evolutionary Search* operator.

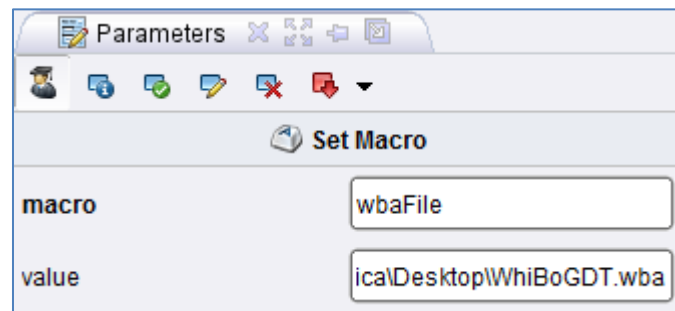


Figure 42 - Parameters panel of Set Macro operator

*GDT Evolutionary Search* operator is set after this. Previously defined Macro is set in *wba file path macro name* parameter text box. Also, log file path is defined. In that file results will be stored. Setup for algorithm search space and parameters of genetic algorithms are below.

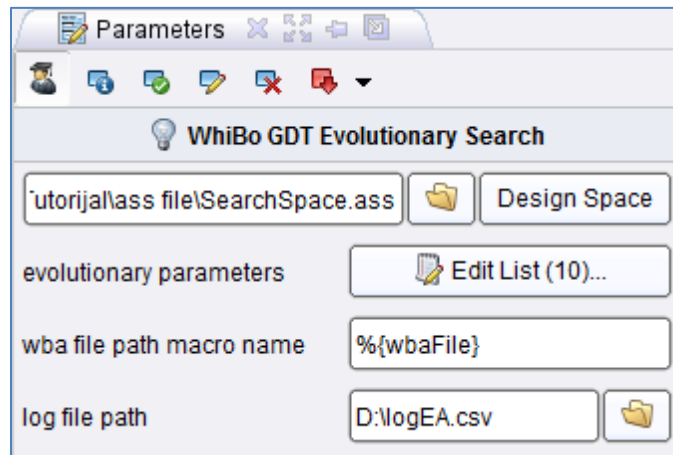


Figure 43 - Parameters panel of *GDT Evolutionary Search* operator

Algorithm search space must be specified.

First, define Create split sub-problem space:

- Click on Create split sub-problem on the left panel.
- Select all components from central panel (multiple components for one sub-problem are selected by holding CTRL key and clicking on components)
- Set default lower and upper values for *Merge\_Alpha\_Value* and *Split\_Alpha\_Value* for *SignificantCategoricalComponent*.
- Click on Save component button from central panel.

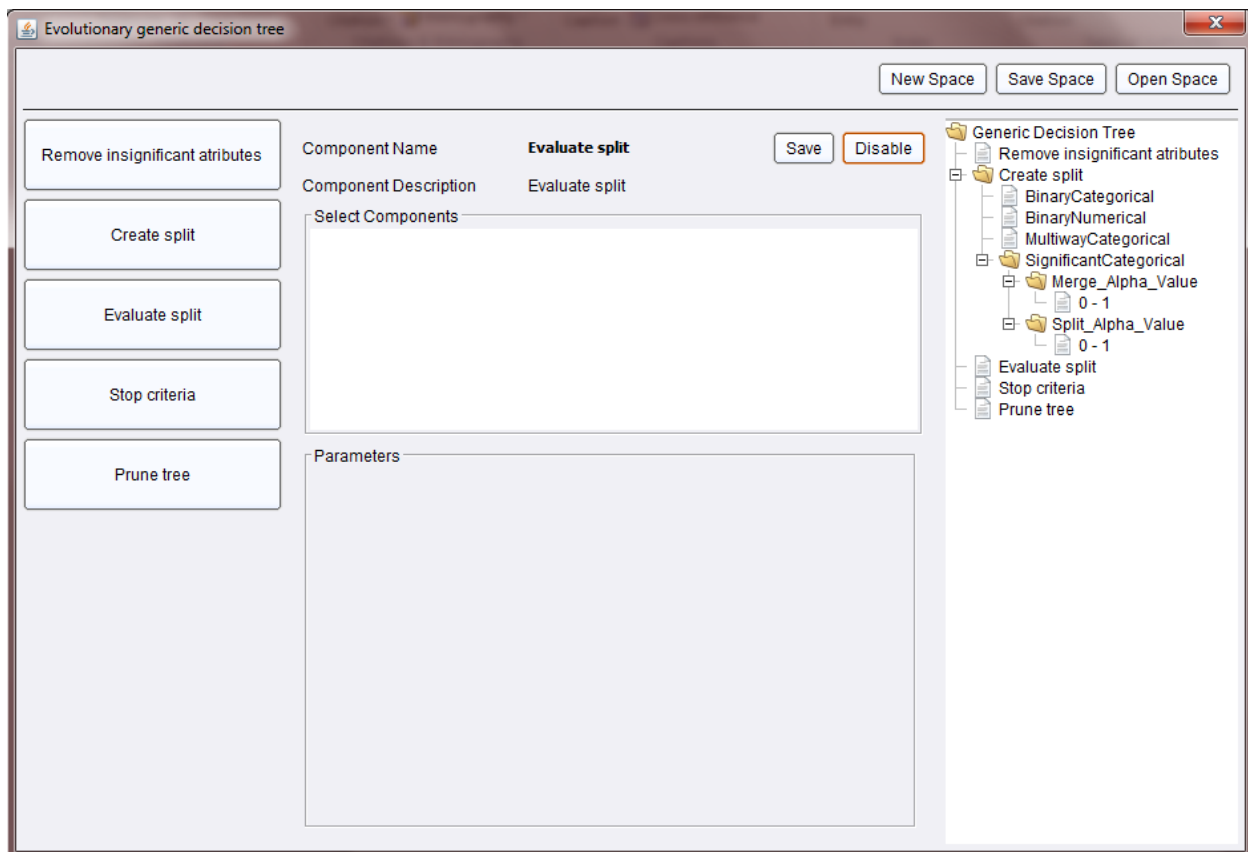


Figure 44 - Definition of *Create split* components for algorithm search space

Second, define Evaluate split sub-problem:

- Click on *Evaluate split* sub-problem on the left panel.
- Select *ChiSquare*, *DistanceMeasure* and *GainRatio* component from central panel.
- Click on save component button from central panel.

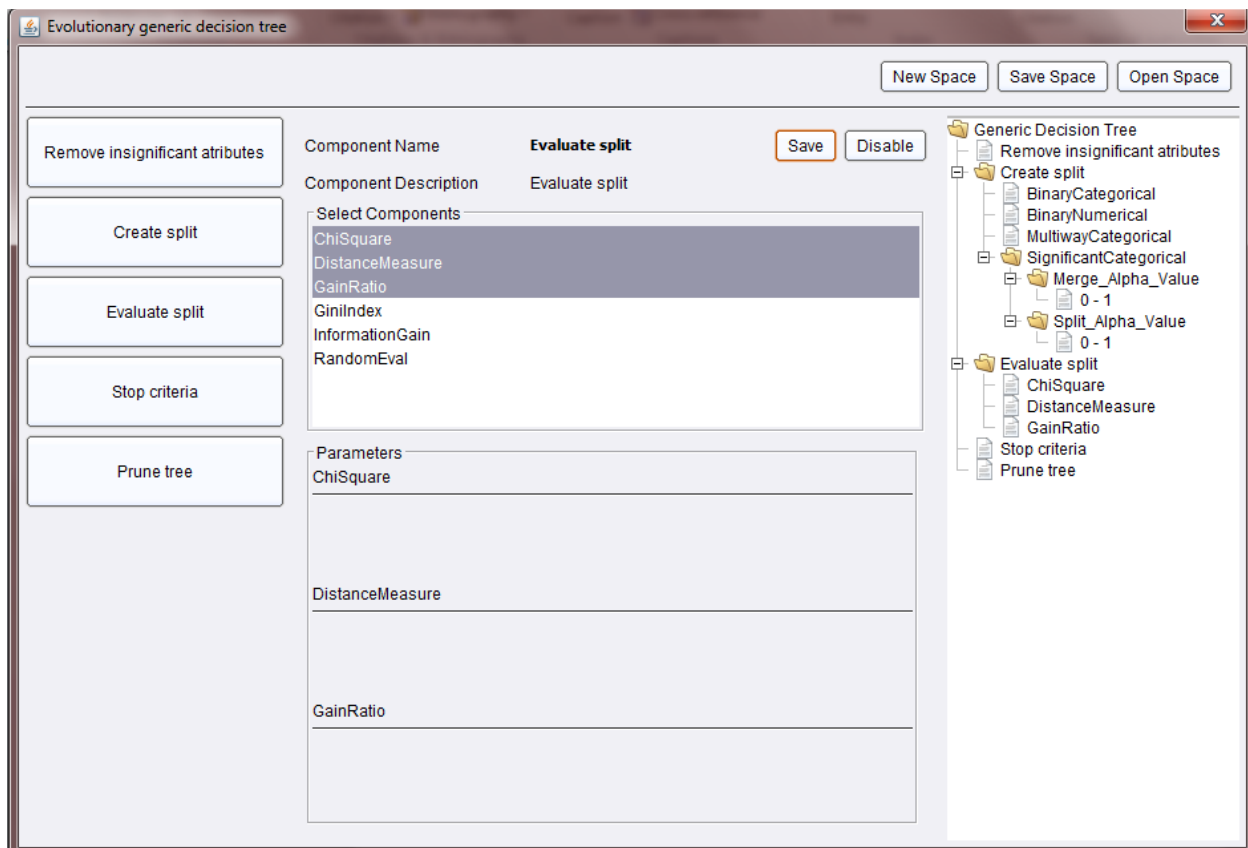


Figure 45 - Definition of *Evaluate split* components for algorithm search space

Third, define *Stop criteria* sub-problem:

- Click on *Stop criteria* sub-problem on the left panel.
- Select *TreeDepth* component from central panel.
- Set *Tree\_Depth* parameter from 1 to 10.
- Click on save component button from central panel.

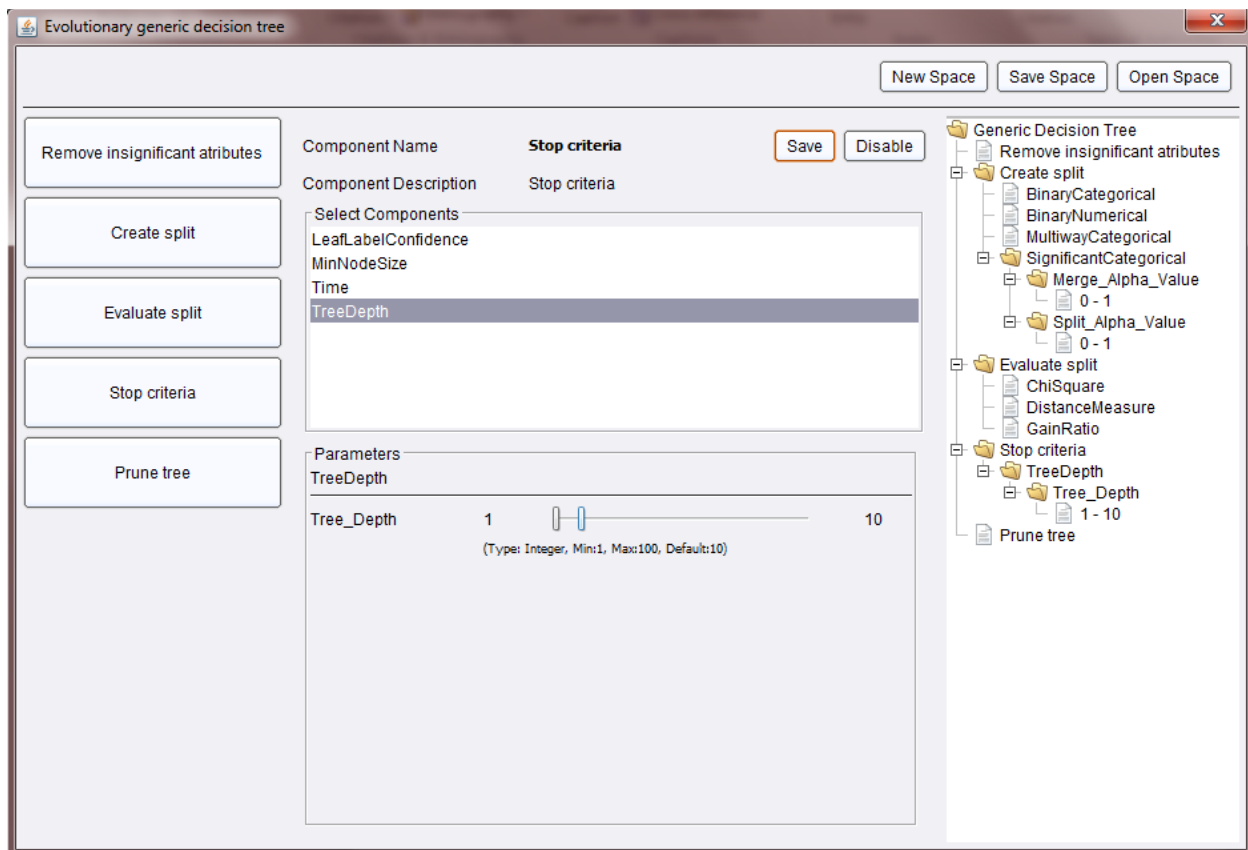


Figure 46 - Definition of *Stop criteria* components for algorithm search space

Fourth, define *Prune tree* sub-problem:

- Click on *Prune tree* sub-problem on the left panel.
- Select *PessimisticError* component from central panel.
- Set *Confidence\_Level* parameter from 0 to 0.5.
- Click on save component button from central panel.

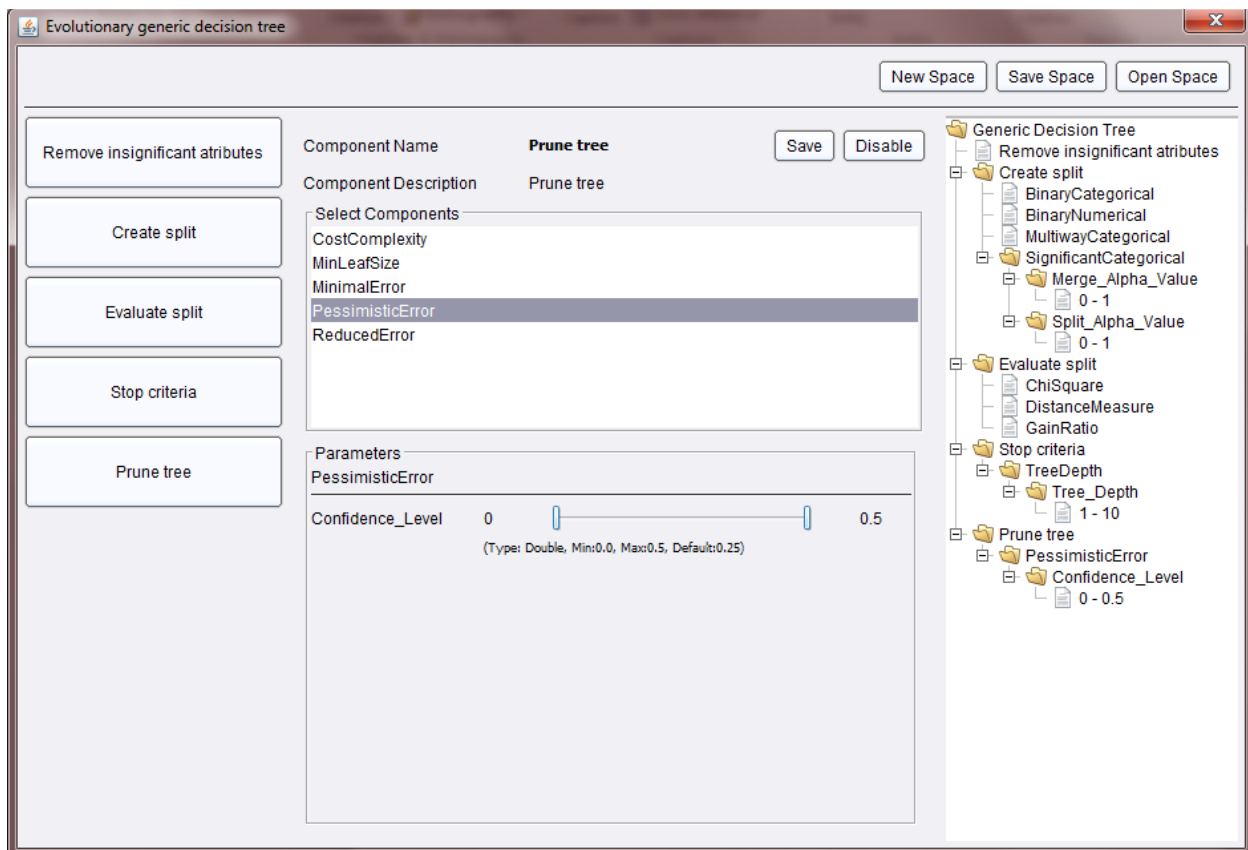
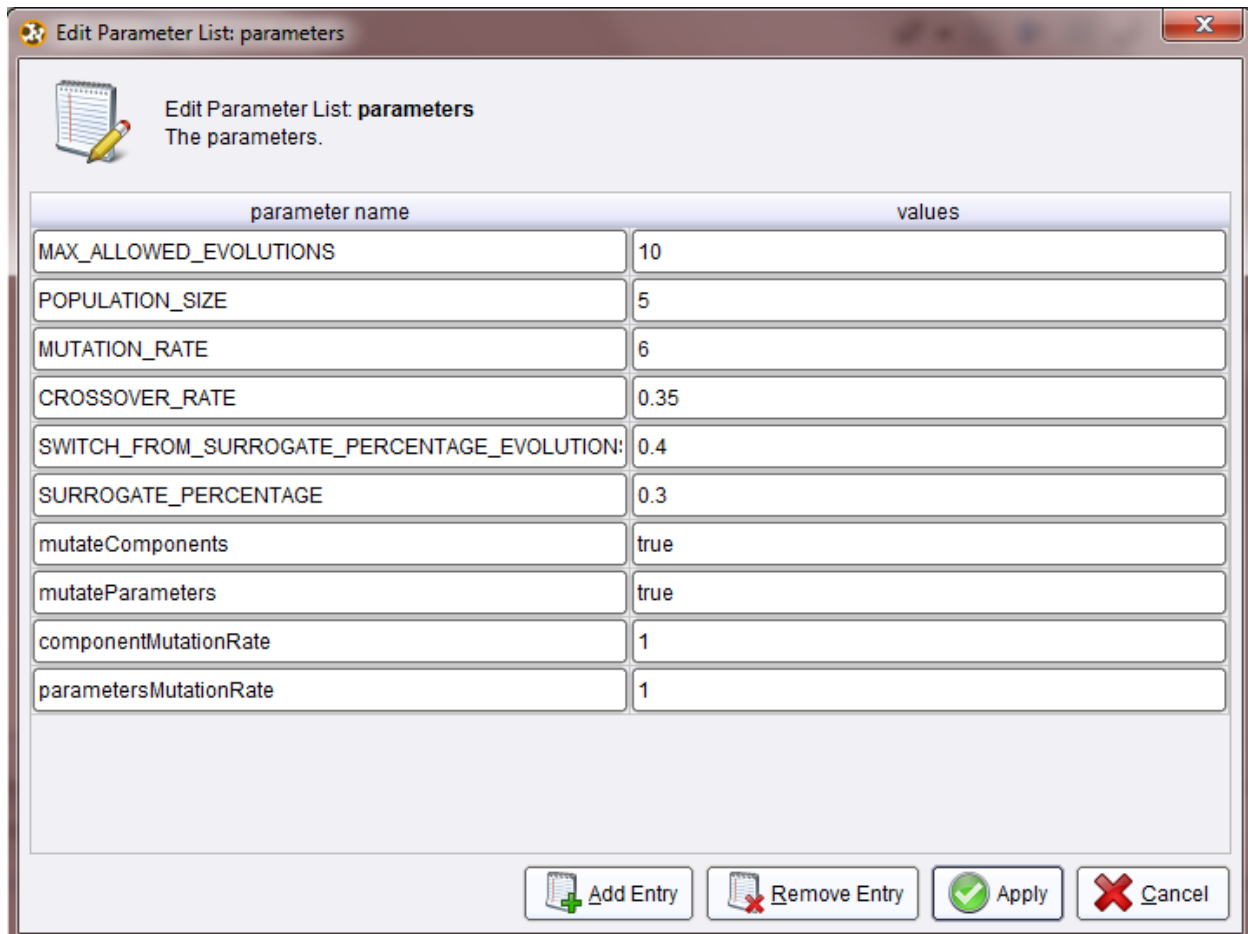


Figure 47 - Definition of *Prune tree* components for algorithm search space

Next step in configuring *GDT Evolutionary Search* operator is parameter settings of genetic algorithm by clicking Edit List button next to parameters in Parameters panel:

- Set MAX\_ALLOWED\_POPULATION to 10.
- Set POPULATION\_SIZE to 5
- Set mutateParameters to true.
- Click on Apply button.



Edit Parameter List: parameters

Edit Parameter List: **parameters**  
The parameters.

| parameter name                              | values |
|---|--------|
| MAX_ALLOWED_EVOLUTIONS                      | 10     |
| POPULATION_SIZE                             | 5      |
| MUTATION_RATE                               | 6      |
| CROSSOVER_RATE                              | 0.35   |
| SWITCH_FROM_SURROGATE_PERCENTAGE_EVOLUTION: | 0.4    |
| SURROGATE_PERCENTAGE                        | 0.3    |
| mutateComponents                            | true   |
| mutateParameters                            | true   |
| componentMutationRate                       | 1      |
| parametersMutationRate                      | 1      |





 Add Entry  Remove Entry  Apply  Cancel

Figure 48 - Definition of parameters for genetic algorithm



Inside *GDT Evolutionary Search* operator is cross validation operator, and inside cross validation there is *GDT* operator in Training section, while *Apply Model* and *Performance* operators are in Testing section. *GDT* should have valid .wba file, with defined *Create split* and *Evaluate split* components.

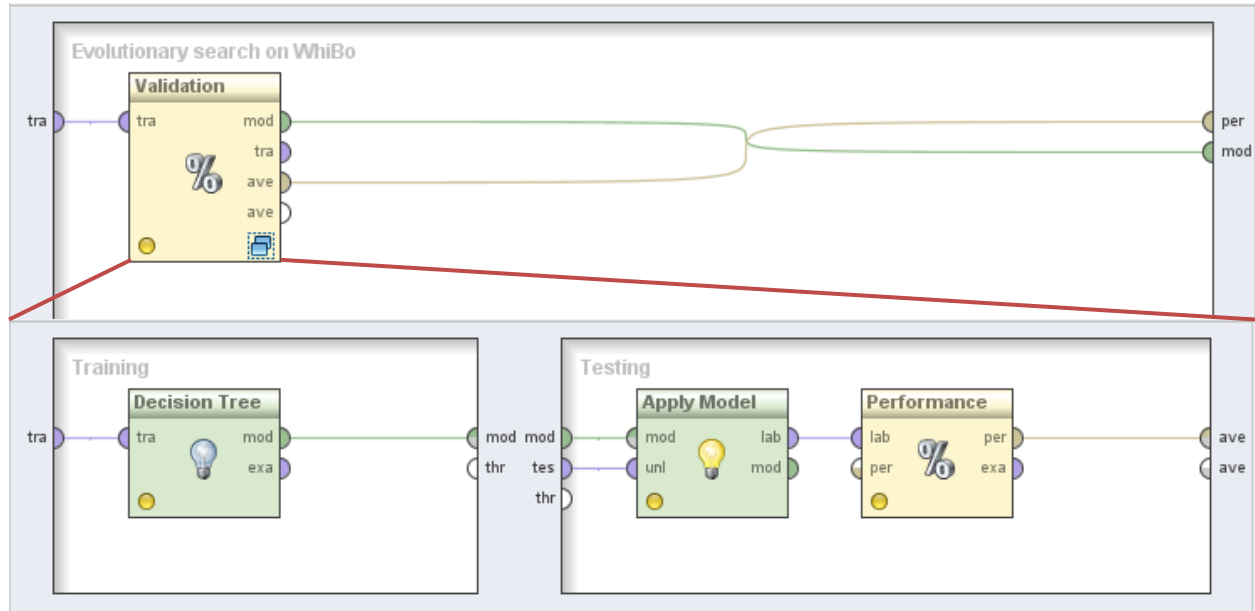


Figure 49 - *GDT Evolutionary Search* subprocess

Result from executed algorithm is shown on figures below.

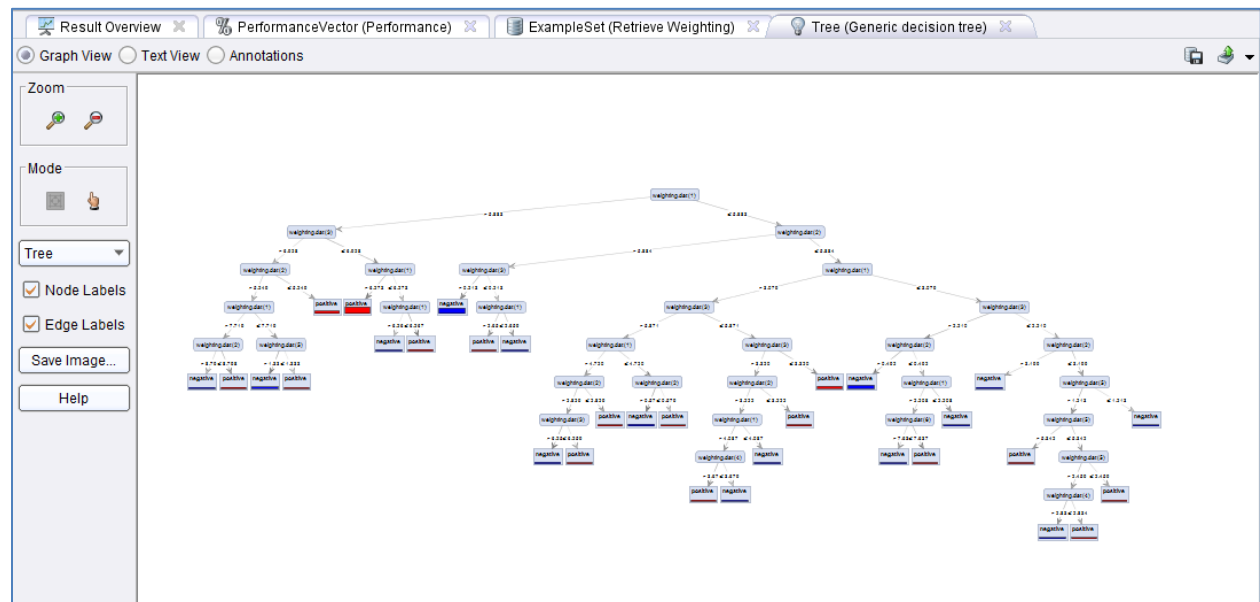


Figure 50 - Result of executed *GDT Evolutionary Search* example

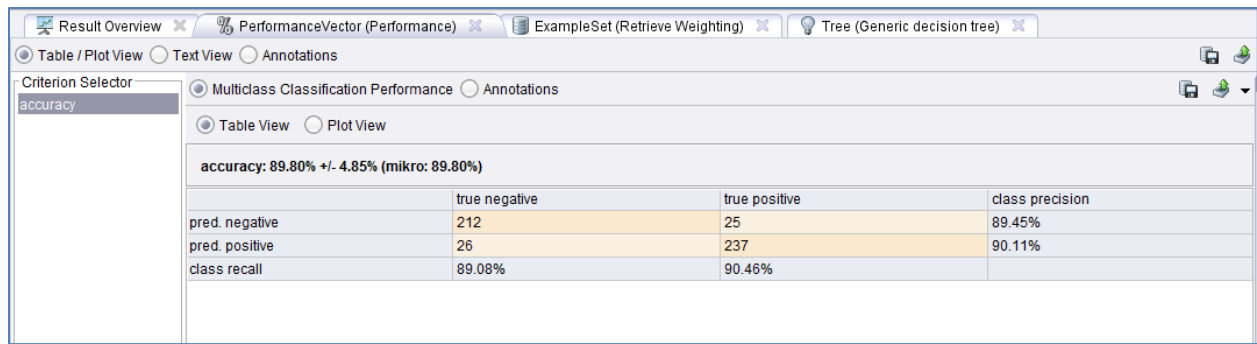


Figure 51 - Performance of executed *GDT Evolutionary Search* example

|    |   |                   |                 |               |   |
|----|---|-------------------|-----------------|---------------|---|
| 21 | Number of values returned from cache: 6       |                   |                 |               |   |
| 22 | Number of evaluations of fitness function: 16 |                   |                 |               |   |
| 23 | Execution time: 00:07                         |                   |                 |               |   |
| 24 | Cache cleared-----                            |                   |                 |               |   |
| 25 | null  | BinaryCategorical | ChiSquare       | TreeDepth(46) | PessimisticError(0.08941713426889725) 0.92  |
| 26 | null  | BinaryCategorical | ChiSquare       | TreeDepth(77) | PessimisticError(0.08941713426889725) 0.91  |
| 27 | null  | BinaryCategorical | DistanceMeasure | TreeDepth(77) | PessimisticError(0.08941713426889725) 0.892 |
| 28 | null  | BinaryCategorical | GainRatio       | TreeDepth(77) | PessimisticError(0.08941713426889725) 0.898 |
| 29 | null  | BinaryCategorical | ChiSquare       | TreeDepth(77) | PessimisticError(0.3948348474061291) 0.912  |
| 30 | null  | BinaryCategorical | ChiSquare       | TreeDepth(15) | PessimisticError(0.3948348474061291) 0.894  |
| 31 | null  | BinaryCategorical | ChiSquare       | TreeDepth(78) | PessimisticError(0.3948348474061291) 0.896  |
| 32 | null  | BinaryCategorical | ChiSquare       | TreeDepth(78) | null 0.908                                  |
| 33 | null  | BinaryCategorical | ChiSquare       | TreeDepth(17) | PessimisticError(0.3948348474061291) 0.902  |
| 34 | null  | BinaryCategorical | ChiSquare       | null          | PessimisticError(0.3948348474061291) 0.902  |
| 35 | The best solution fitness value:              | 0.92              |                 |               |   |
| 36 | Best Solution:                                |                   |                 |               |   |
| 37 | null  | BinaryCategorical | ChiSquare       | TreeDepth(17) | PessimisticError(0.3948348474061291)        |
| 38 | Number of values returned from cache: 25      |                   |                 |               |   |
| 39 | Number of evaluations of fitness function: 10 |                   |                 |               |   |

Figure 52 - Log file of executed *GDT Evolutionary Search* example

## Modifying algorithm search space

Modifying of existing algorithm search space is done the same way as creating of new algorithms, by clicking Design algorithm button. In this example we will show how to

- modify parameter of already defined component,
- change component of defined sub-problem,
- adding new sub-problem and component for its solution.

## Modifying parameters

In this example we will modify *Merge\_Alpha\_Value* parameters of *Create split* – *SignificantCategorical* component:

- Click on *Create split* sub-problem on the left panel.
- Set *Merge\_Alpha\_Value* from 0.1 to 0.4.
- Click on save component button from central panel.
- Save algorithm search space.

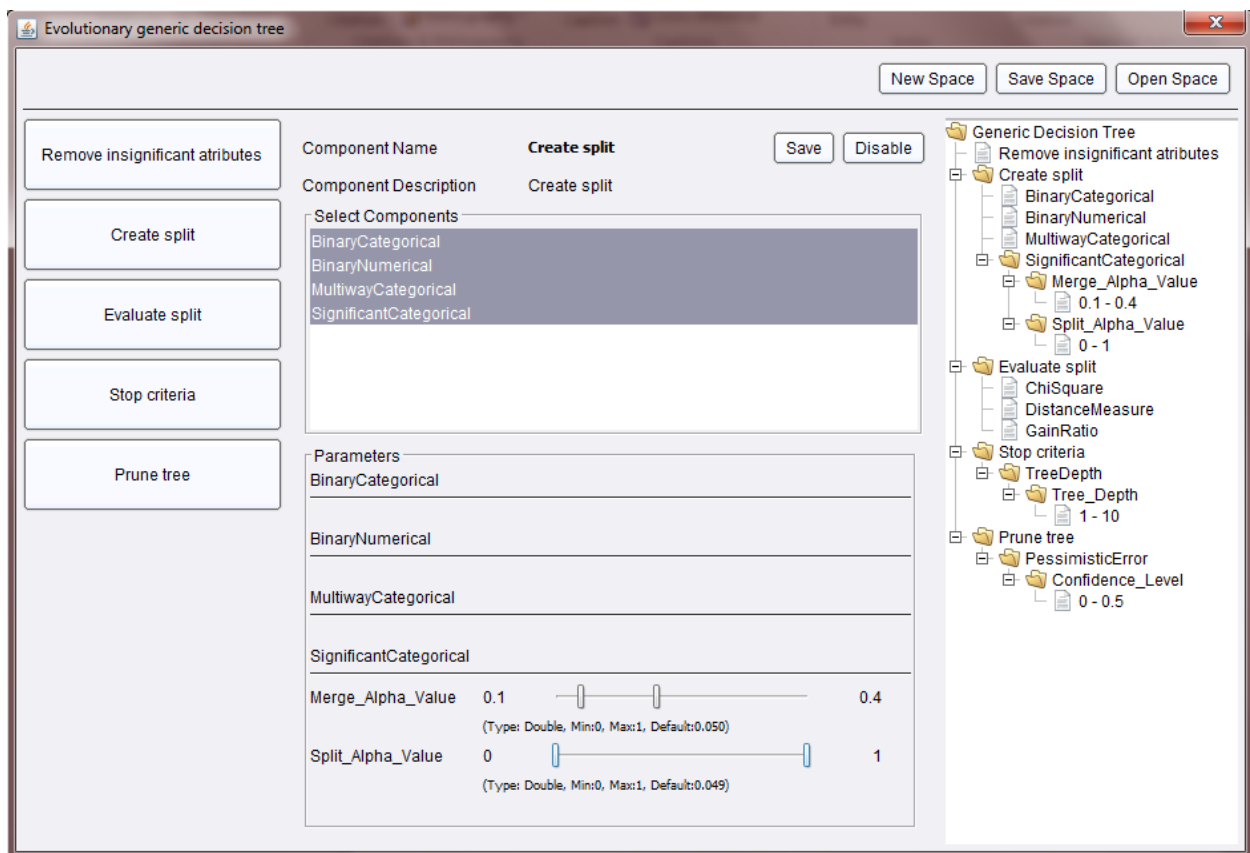


Figure 53 - Modifying parameters of *Merge\_Alpha\_Value*

## Replacing components for sub-problem

Replacing components for sub-problem is done in following way:

- Click on *Evaluate split* sub-problem on the left panel.
- Add InformationGain component (using CTRL button) from central panel.
- Remove GainRatio component (using CTRL button).
- Click on save component button from central panel.
- Save algorithm search space.

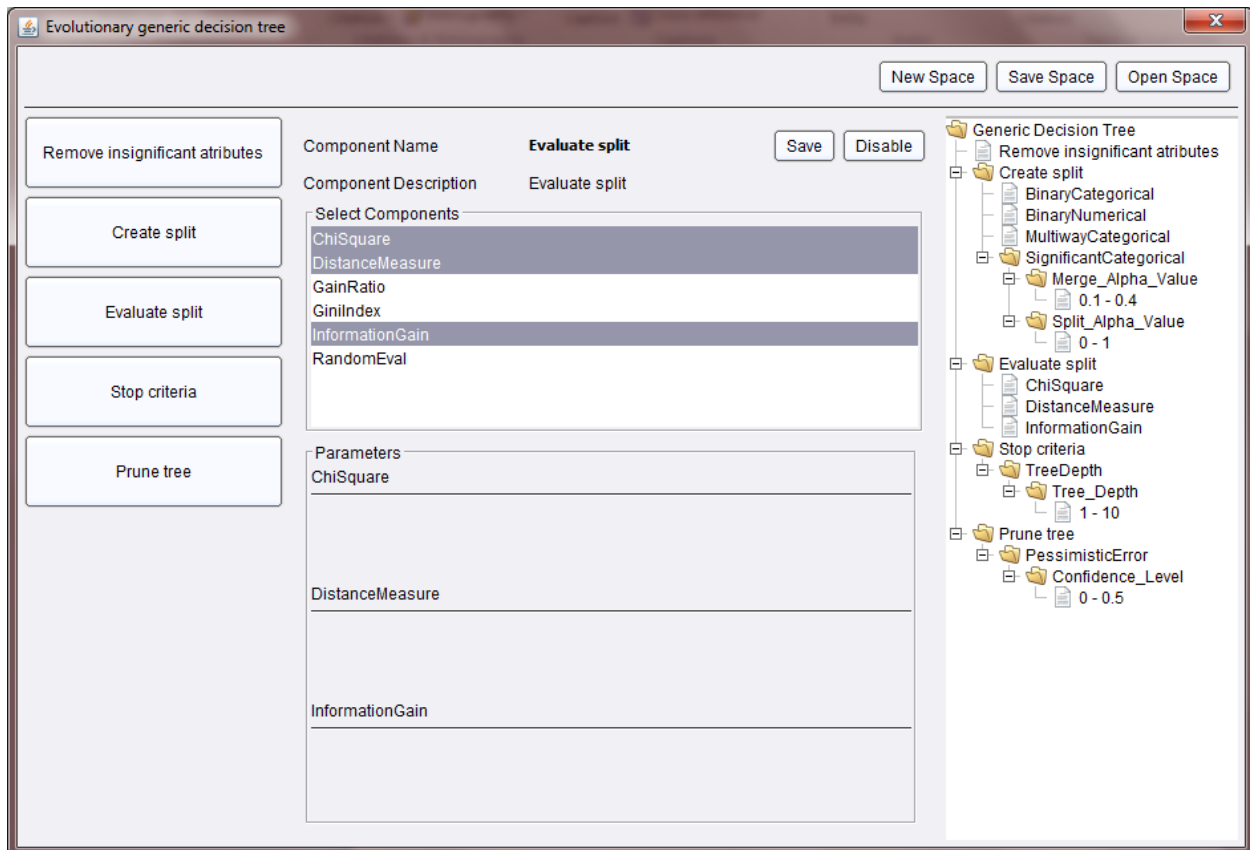


Figure 54 - Replacing components in *GDT Evolutionary Search*

## Adding new sub-problem and component to an existing algorithm

Besides changing of parameters and components existing algorithm search space could be extended with new sub-problems and components. We will explain this extension by adding Remove insignificant attributes sub-problem.

- Select *Remove insignificant attributes* sub-problem from left panel.
- Select *FTestNumerical* component from central panel.
- Leave default parameters setting.
- Click on save component button from central panel.
- Save algorithm search space.

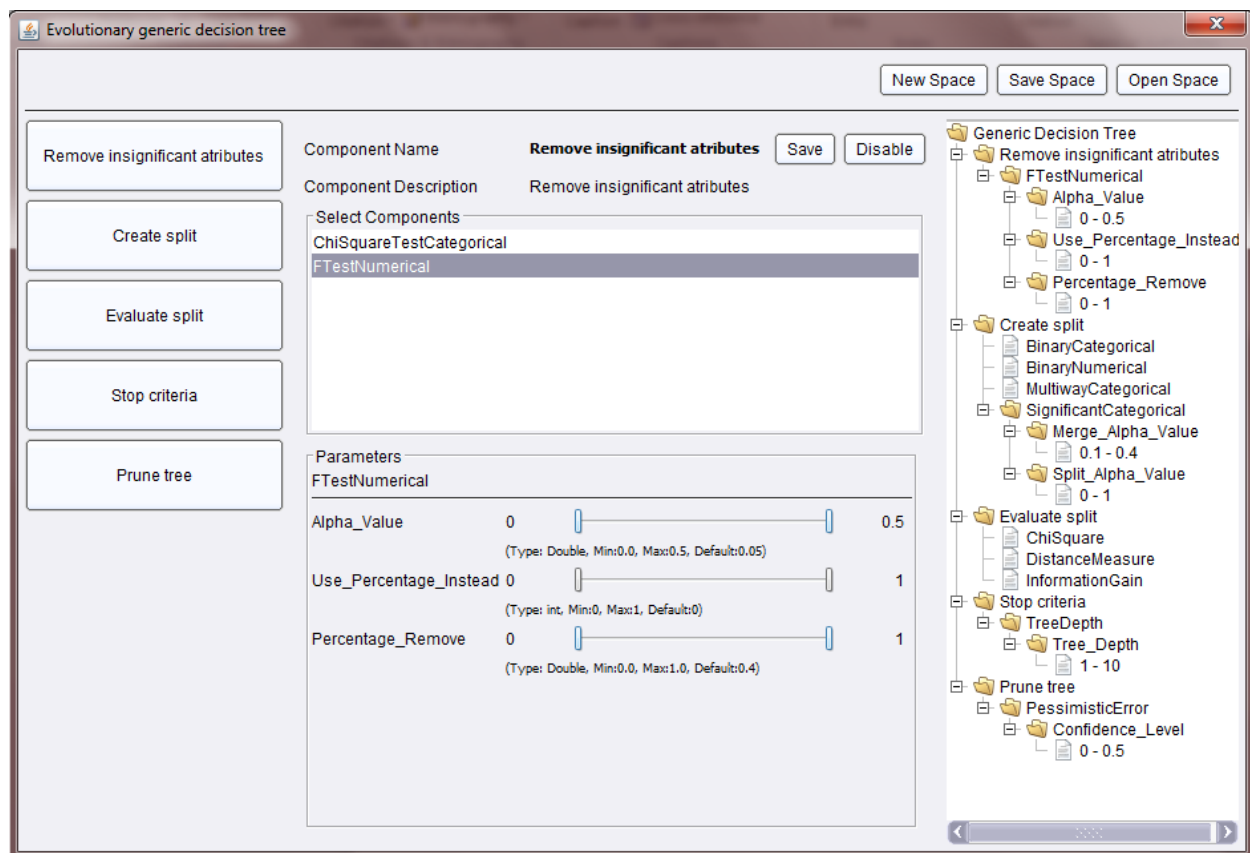


Figure 55 - Adding new subproblem in *GDTEvolutionary Search*

After conducting experiment with this algorithm search space setup results shown on figures below were gathered.

|  |               |               |                 |  |
|--|---------------|---------------|-----------------|--|
| Result Overview                            |               |               |                 |  |
| PerformanceVector (Performance)            |               |               |                 |  |
| ExampleSet (Retrieve Weighting)            |               |               |                 |  |
| Tree (Generic decision tree)               |               |               |                 |  |
| Table / Plot View                          |               |               |                 |  |
| Text View                                  |               |               |                 |  |
| Annotations                                |               |               |                 |  |
| Criterion Selector                         |               |               |                 |  |
| accuracy                                   |               |               |                 |  |
| Multiclass Classification Performance      |               |               |                 |  |
| Annotations                                |               |               |                 |  |
| Table View                                 |               |               |                 |  |
| Plot View                                  |               |               |                 |  |
| accuracy: 91.00% +/- 2.72% (mikro: 91.00%) |               |               |                 |  |
| pred. negative                             | true negative | true positive | class precision |  |
|  | 213           | 20            | 91.42%          |  |
| pred. positive                             | 25            | 242           | 90.64%          |  |
| class recall                               | 89.50%        | 92.37%        |                 |  |

Figure 56 - Performance of executed *GDT Evolutionary Search* example

|    |   |                     |                 |      |                                       |       |
|----|---|---------------------|-----------------|------|---------------------------------------|-------|
| 54 | Cache cleared-----                            |                     |                 |      |                                       |       |
| 55 | Number of values returned from cache: 13      |                     |                 |      |                                       |       |
| 56 | Number of evaluations of fitness function: 12 |                     |                 |      |                                       |       |
| 57 | Execution time: 00:04                         |                     |                 |      |                                       |       |
| 58 | Cache cleared-----                            |                     |                 |      |                                       |       |
| 59 | FTestNumerical(0.9729494240097137)            | MultiwayCategorical | DistanceMeasure | null | PessimisticError(0.43504839077645285) | 0.898 |
| 60 | FTestNumerical(0.5242083226527987)            | MultiwayCategorical | DistanceMeasure | null | PessimisticError(0.291387730739167)   | 0.912 |
| 61 | FTestNumerical(0.27080173073754676)           | MultiwayCategorical | DistanceMeasure | null | PessimisticError(0.291387730739167)   | 0.902 |
| 62 | FTestNumerical(0.9053335432718563)            | MultiwayCategorical | DistanceMeasure | null | PessimisticError(0.2137296357883106)  | 0.904 |
| 63 | FTestNumerical(0.34203456281548106)           | MultiwayCategorical | DistanceMeasure | null | PessimisticError(0.2137296357883106)  | 0.912 |
| 64 | FTestNumerical(0.662099165297556)             | MultiwayCategorical | DistanceMeasure | null | PessimisticError(0.2137296357883106)  | 0.898 |
| 65 | The best solution fitness value:              | 0.912               |                 |      |                                       |       |
| 66 | Best Solution:                                |                     |                 |      |                                       |       |
| 67 | FTestNumerical(0.34203456281548106)           | MultiwayCategorical | DistanceMeasure | null | PessimisticError(0.2137296357883106)  |       |
| 68 | Number of values returned from cache: 26      |                     |                 |      |                                       |       |
| 69 | Number of evaluations of fitness function: 6  |                     |                 |      |                                       |       |

Figure 57 - Log file of executed *GDT Evolutionary Search* example

## Extending WHIBO

Input and output are well defined for every Sub-problem, and these sub-problems are implemented as abstract classes in WhiBo. Reusable components are concrete classes where the logic is implemented. Sub-problems define standardized input and output for every reusable component, extended from sub-problem.

WhiBo is implemented as an extendable environment in the Java programming language that enables the implementation of new RCs and sub-problems. Extending the GDT can be done by:

- Adding new RCs.
- Adding new sub-problems in the existing GDT algorithm.

The GDT algorithm is implemented independently of RCs. So extending the GDT algorithm with a new RC asks for no changes in the algorithm flow. On the other hand, when extending WhiBo with a new sub-problem changes are needed in the GDT algorithm.

Adding a new RC is accomplished in two steps. The first step is to define a new class for the RC. For that class the user has to define parameters and implement the RC logic. The inputs and outputs of the RC are predefined by the sub-problem the RC belongs to. The necessary changes are shown in bold at Figure 55. The second step is to register the new RC for a sub-problem as shown in bold at Figure 56.

If these two steps are done correctly the user will see his own component in the central panel of WhiBo GUI (Figure 4), and can use the GDT with the new RC.

Adding a new sub-problem is achieved in three steps. The first step is to create an interface for the sub-problem, and define inputs and outputs for the sub-problem as shown in bold at Figure 57.

The second step is to register the new sub-problem to enable using it through GUI as shown in bold at Figure 58.

Finally, the user has to modify the existing GDT algorithm to utilize the newly defined sub-problem. WhiBo is not only intended for use with decision-tree algorithms, but can be extended to other component-based machine learning algorithms.

```
package rs.fon.WhiBo.GDT.component.splitEvaluation;
public class MySplitEvaluation
    extends AbstractSplitEvaluation {

    @Parameter(defaultValue="0.05", minValue ="0",
                maxValue="1")
        private Double Alpha_Value;

    @Override
        public double evaluate(SplittedExampleSet
exampleSet)
        {
            /*
                user implementation
                for candidate split evaluation
            */
            return splitEvaluation;
        }
}
```

Figure 58 - Implementing a new RC

```
public class SplitEvaluation implements Subproblem {
    ...
    PrivateString[] availableImplementationClassNames
=
    {
        GainRatio.class.getName(),
        GiniIndex.class.getName(),
        InformationGain.class.getName(),
        DistanceMeasure.class.getName(),
        ChiSquare_FTest.class.getName(),
        MySplitEvaluation.class.getName();
    }
};
```

Figure 59 - Registering the new RC for a sub-problem



```

package
rs.fon.WhiBo.GDT.component.newSubproblem;
public interface newSubproblem {
    public output1 newSubproblemMethod1(inputs1);
    public output2 newSubproblemMethod1(inputs2);
}

```

Figure 60 - Defining a new sub-problem

WhiBo can be found at the following web page <http://code.google.com/p/WhiBo/>. Data mining and machine learning researchers are invited to join our efforts to exchange components of decision trees and other machine learning algorithms in an open way based on the proposed WhiBo platform, as to establish a standard for interchange of components among decision tree based classification algorithms, as well as other machine learning algorithms.

```

Package rs.fon.WhiBo.GDT.problem;
....
public class GenericTreeProblemBuilder {

    public Problem buildProcess() {
        ...
        Subproblem s2 = new PossibleSplit();
        Subproblem s3 = new Split Evaluation();
        ...
        Subproblem s7 = new UserDefinedSubproblem();
        List<Subproblem> subproblems;
        subproblems.add(s1);
        subproblems.add(s2);
        ...
        subproblems.add(s7);
        Problem process = new GenericTreeProblem();
        process.setProcessSteps(steps);
        return process;
    }
....
}

```

Figure 61 - Registering the new sub-problem

## Developer guide

In order to extend WhiBo there are several steps which needs to be done.

1. Since WhiBo is written in Java programming language, first step is to download Eclipse (<http://www.eclipse.org/downloads/>).
2. When Eclipse is downloaded subversion support needs to be installed. We recommend **Subclipse**, which can be found on <http://subclipse.tigris.org/servlets/ProjectProcess?pageID=p4wYuA>. Installation of **Subclipse** is done in several steps:
  1. Open **Eclipse**.
  2. Select the **Help > Install New Software** menu option.

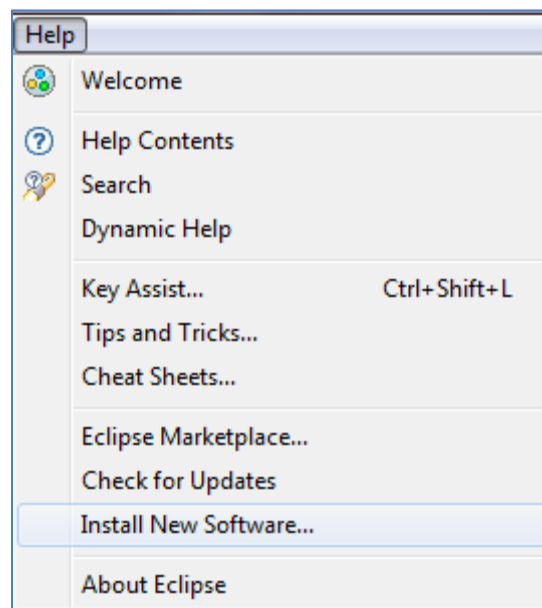


Figure 62 - Installation of Subclipse

3. Click the **Add** button and set the **Location** field on [http://subclipse.tigris.org/update\\_1.8.x](http://subclipse.tigris.org/update_1.8.x), and set name for example Subclipse. Then click **OK** button.

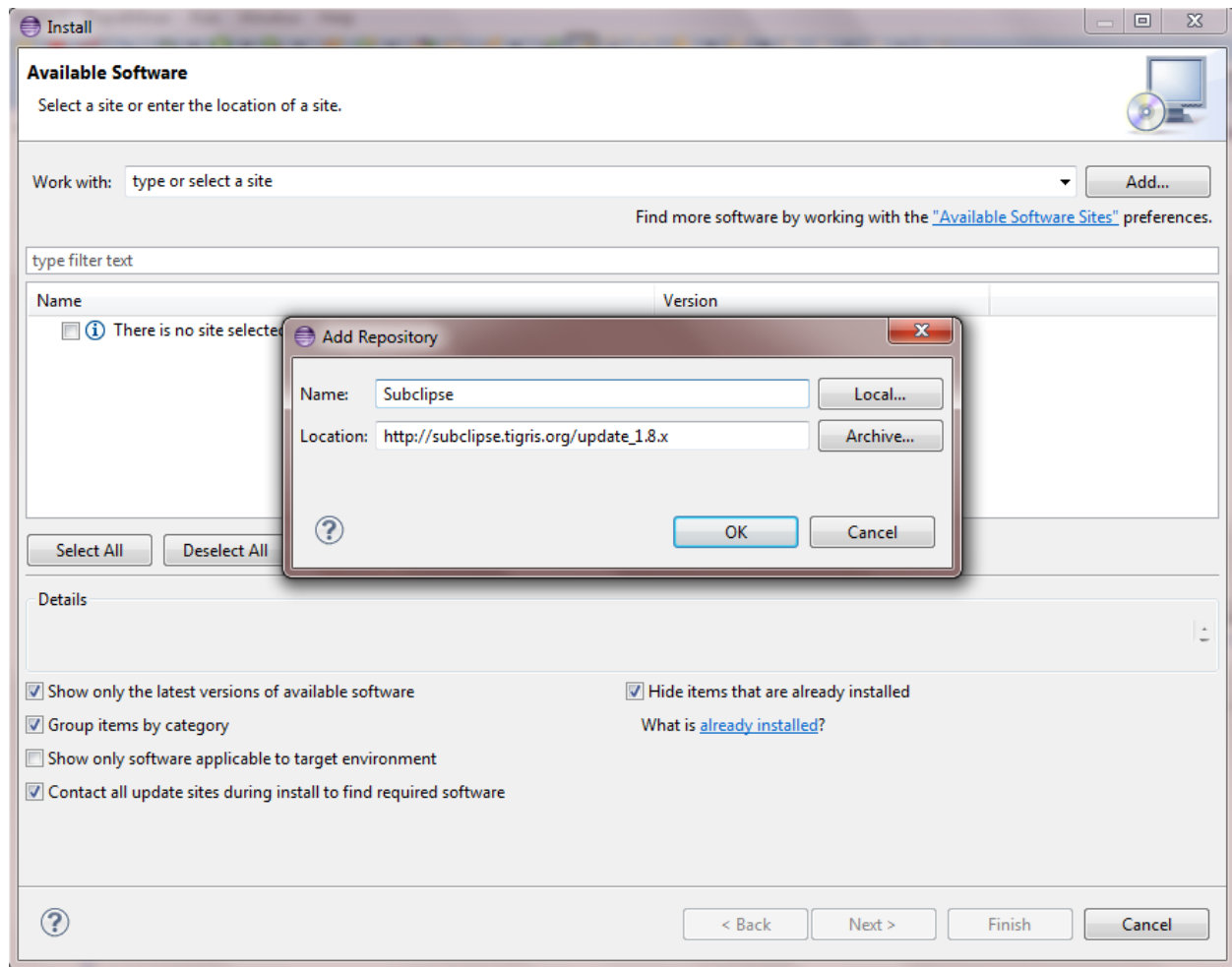


Figure 63 - Adding Subclipse repository

4. Select **Subclipse** components and click **Next**.
5. Select the **I accept the terms of the license agreements** radio button.
6. Click the **Finish** button.
7. Click **Yes** to restart Eclipse.

Eclipse will now have SVN Repository Exploring panel. If Eclipse don't show this panel at first it can be added by clicking **Windows->Open Perspective->Other...**, then selecting **SVN Repository Exploring** option and click OK.

3. Checkout of **WhiBo** project is done in several steps:

1. Right Click a repository in the **SVN Repositories** panel, select **New**, then **Repository location....**

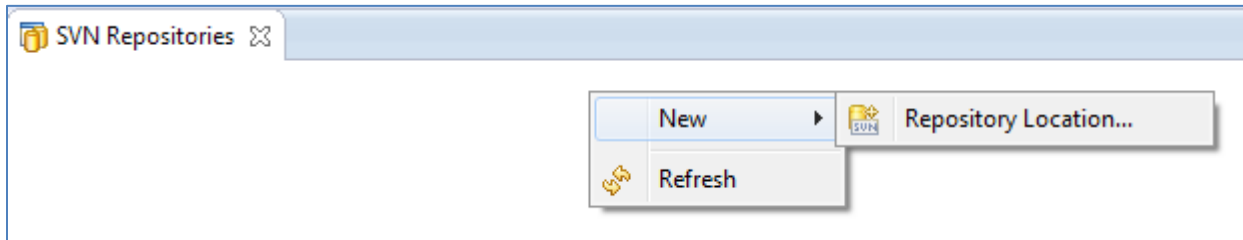


Figure 64 - Adding new repository location

2. Insert <https://whibo.googlecode.com/svn/trunk/> in **URL** text box.

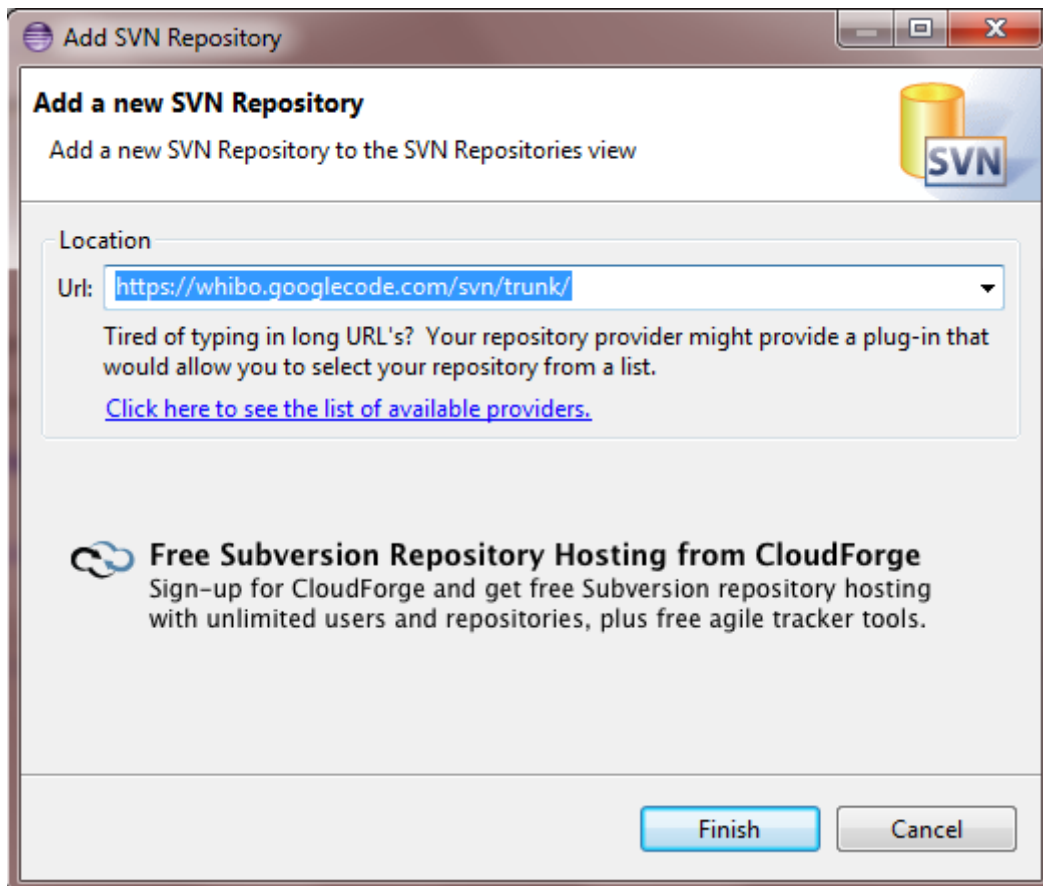


Figure 65 - Adding WhiBo repository location

3. Click **Finish** button.

4. Right click on **WhiBo** repository in **SVN Repositories** panel.
5. Select the **Checkout...** option.

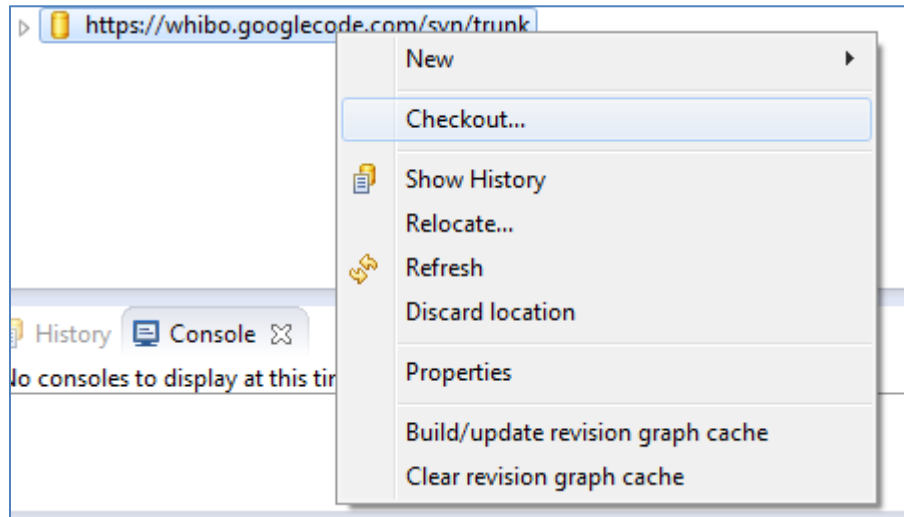


Figure 66 – Checkout of WhiBo project (1)

6. Select the **Check out as a project in the workspace** option and enter a project name.

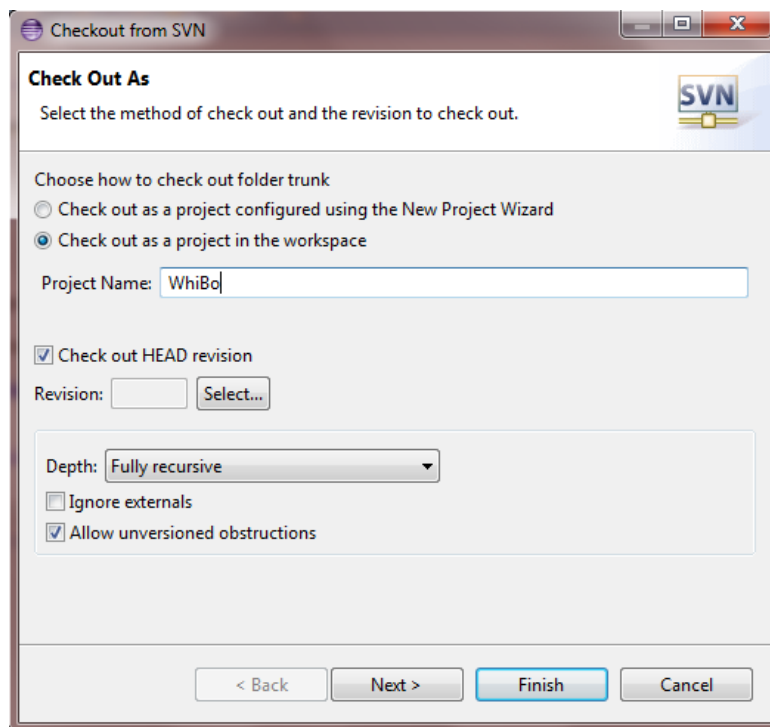


Figure 67 – Checkout of WhiBo project (2)

7. Select workspace where you wish to save project.

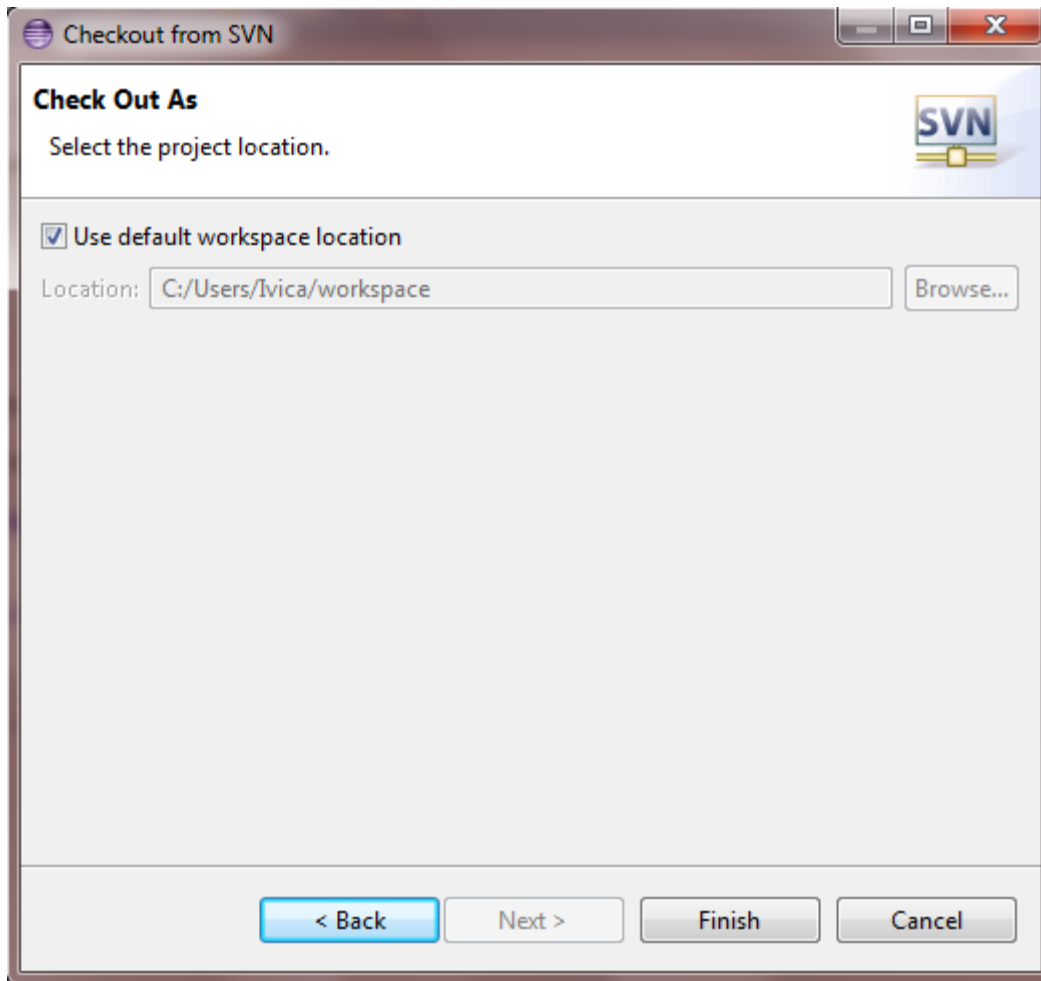


Figure 68 - Selecting workspace location

8. Click **Finish** button.

9. **WhiBo** project will show up in **Package Explorer** panel.

4. Similarly, **RapidMiner** project needs to be imported as project. URL for **RapidMiner** project is <http://svn.code.sf.net/p/rapidminer/code>. Currently, **RapidMiner** version is called **Unuk**.

5. After importing **RapidMiner** project it needs to be referenced in **WhiBo** project.

1. Right click on **WhiBo** project.

2. Click **Properties**.
3. Select **Java Build Path** on left side and then **Project** tab on central panel.

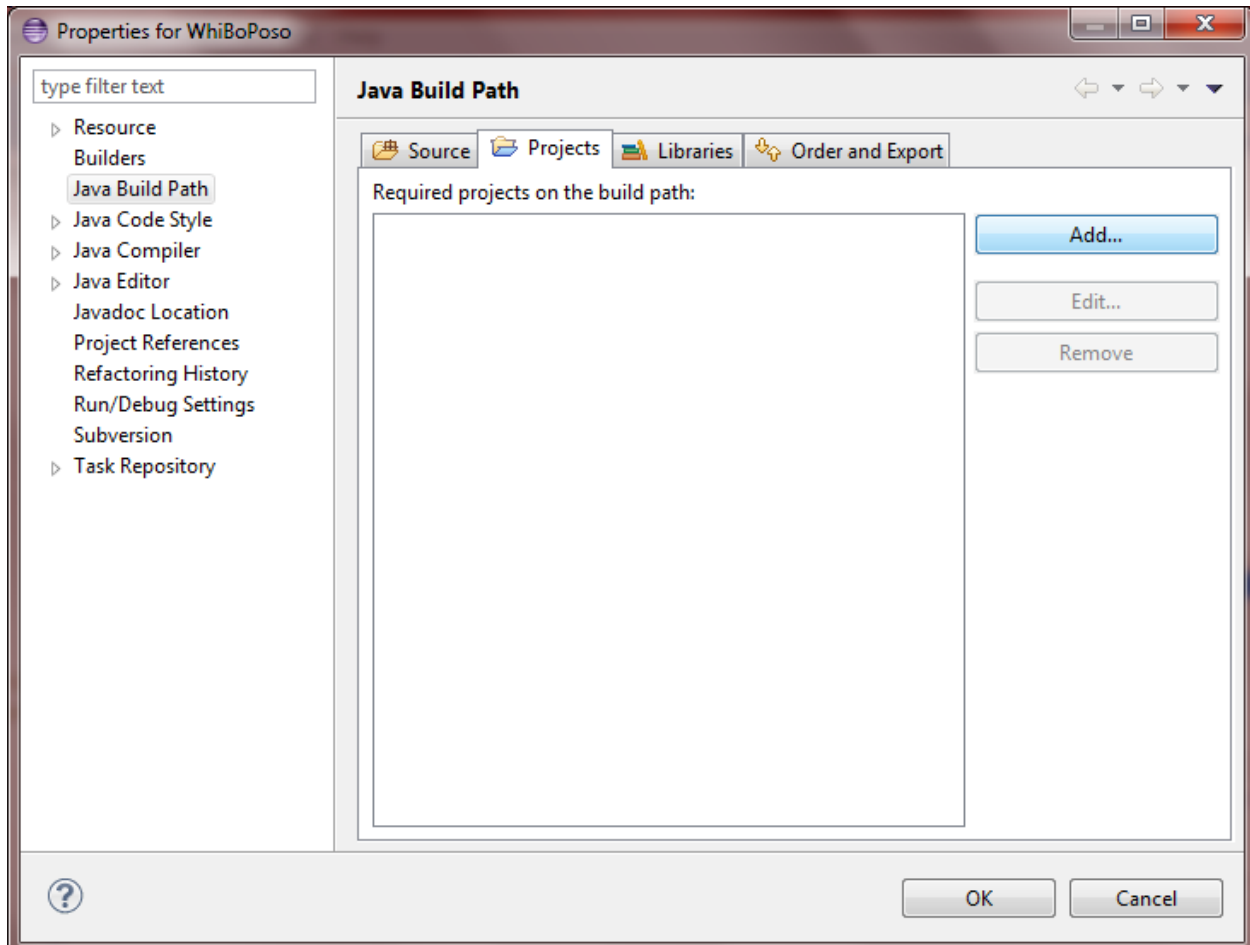


Figure 69 - Importing RapidMiner project into WhiBo project

4. Click **Add...** button.
5. Select proper **RapidMiner** version.
6. Click **OK** button on **Project Selection** panel.
7. Click **OK** button on **Properties** panel.

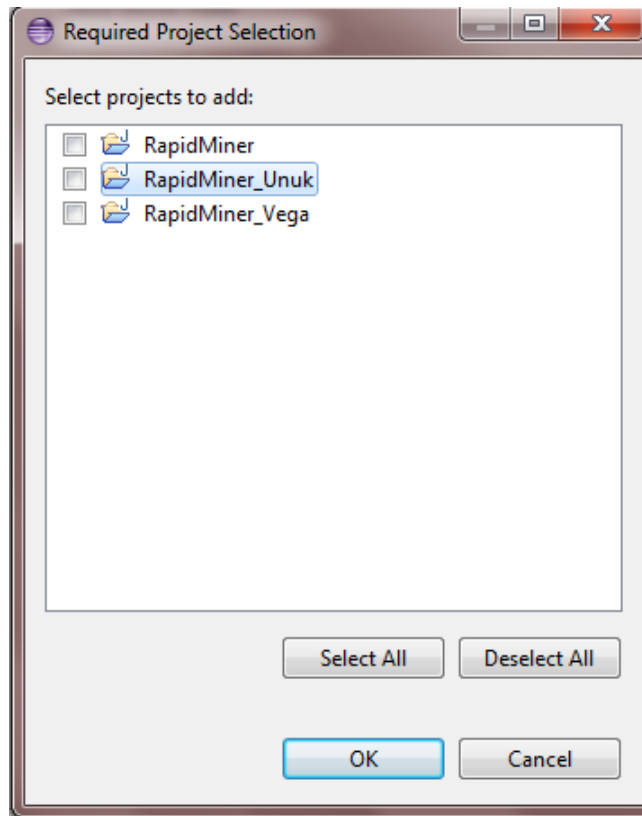


Figure 70 - Selecting RapidMiner version

6. Open **build.xml** file of **WhiBo** project.
7. Make sure that fifth line contains proper **RapidMiner** project (in this case it should be:  
`<property name="rm.dir" location="../../RapidMiner_Unuk" />`)
8. Right click on **build.xml** file and select **Run as...->Ant Build**. With this step **WhiBo** extension is building in **RapidMiner** project, so it can be used in that project.

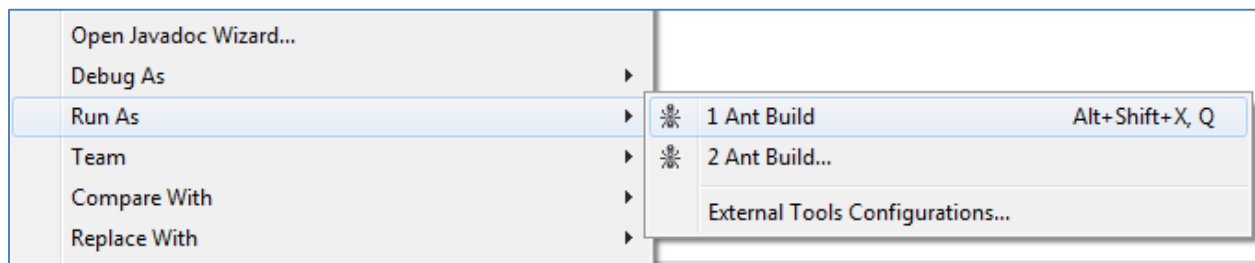


Figure 71 - Building WhiBo project



9. Right click on **WhiBo** project and select **Run as...->Java Application**.

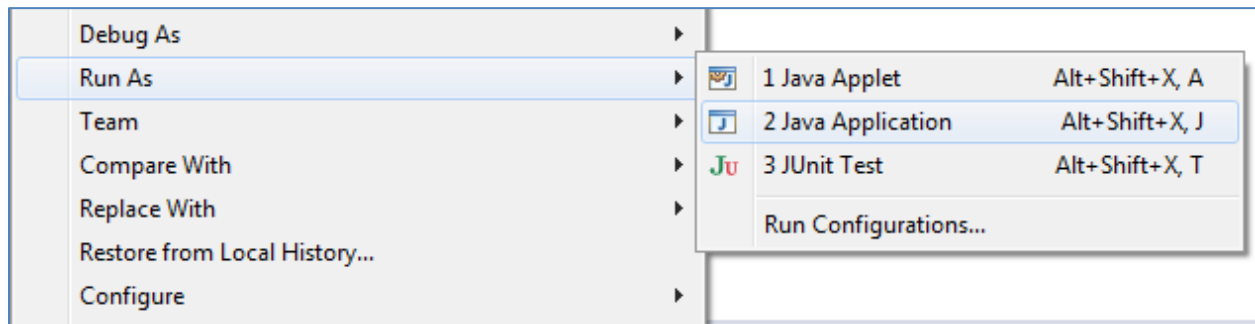


Figure 72 - Running WhiBo project

10. Select **RapidMinerGUI** class.

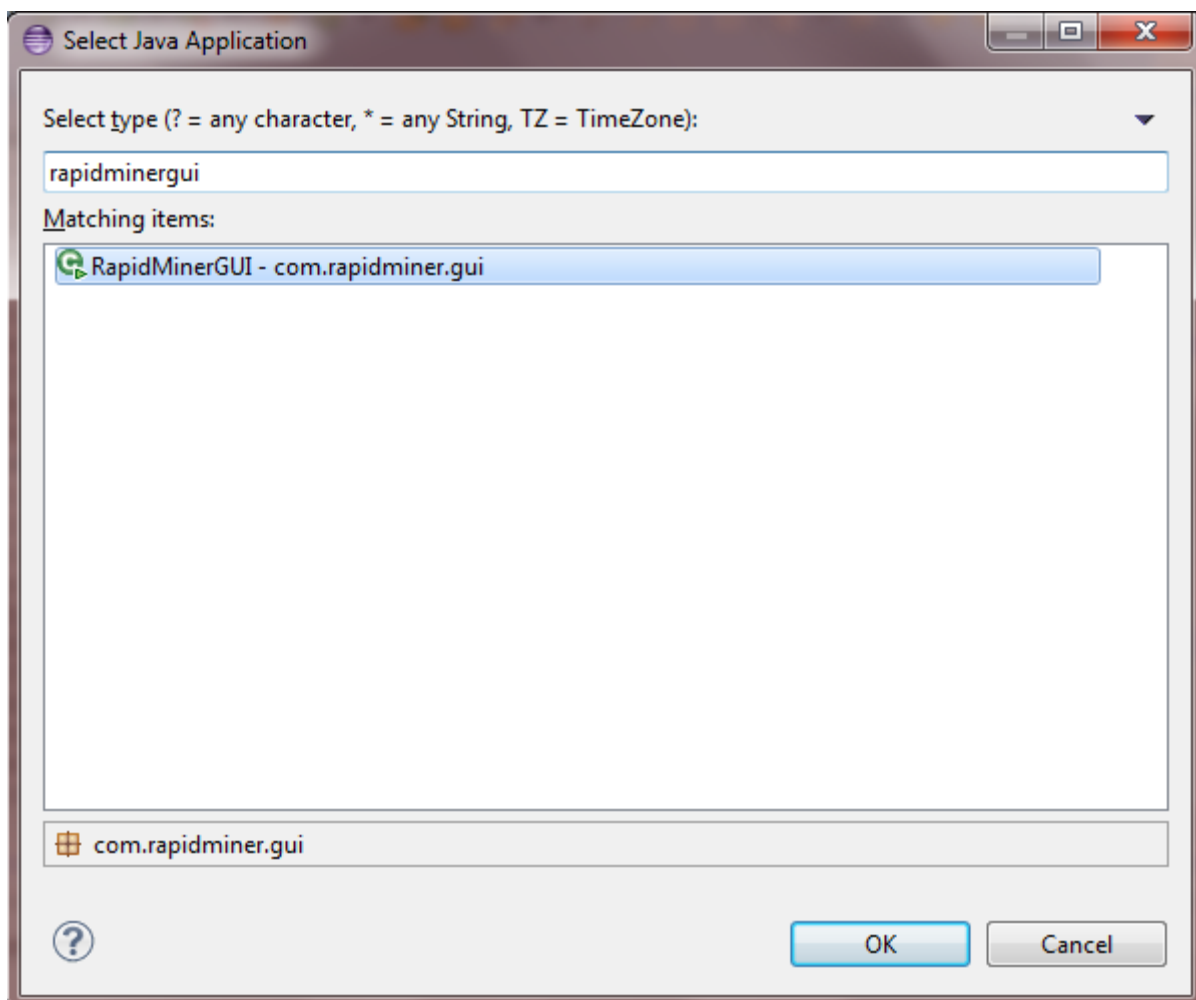


Figure 73 - Main class of WhiBo project

11. **RapidMiner** will start and **WhiBo** can be used.

For any information about configuration and extending WhiBo project you can contact us on e-mails (which can be found on the [website](#)) or on forum (which is also on the [website](#)).

## Appendix A

### ID3 algorithm

This algorithm is the first algorithm of Ross Quinlan (Quinlan, 1986). It can only work with categorical data. It uses information gain as a measure of split quality. This evaluation measure is biased towards choosing attributes with more categories.

(Quinlan 1993).

### CART algorithm

This algorithm is proposed by Breiman et al, 1984. It is a classification and regression tree which can work with both numerical and categorical data. We analyzed only the classification tree.

CART uses for split evaluation three evaluation measures: Gini, Twoing and Ordered Twoing for ordered categorical data. We analyzed only the Gini evaluation measure which is the most frequently used measure.

CART produces only binary splits for categorical, as well as numerical data. CART includes an algorithm for tree pruning, namely cost complexity pruning (CCP). The generated CART model has the option to classify cases with missing attribute values. This is achieved through surrogate splits, i.e. alternative split nodes that are generated during tree growth and should be used as a replacement when the original attribute value is missing.

### C4.5 algorithm

The successor of ID3 algorithm (Quinlan, 1993) improves several aspects of the original tree. It can work with numerical and categorical data. It produces multiway splits for categorical data, and binary splits for numerical data.

It uses a less-biased split evaluation measure, the gain ratio. It includes options to handle missing values (which we didn't analyze), and three pruning algorithms (reduced-error pruning, pessimistic-error pruning and error-based pruning).

## CHAID algorithm

It was proposed by Kass, 1980. It uses the chi-square test to evaluate the quality of a split. It works only with categorical attributes. Instead of branching a node on all categories or binary, it tries to group similar categories in joint categories, merging statistically significant categories together. It produces branches based on these merged categories.

## Distance measure

This split evaluation measure was proposed by (Mantaras, 1991). It is an unbiased multiway evaluation measure. It is an improvement of information gain and gain ratio (although gain ratio was proposed later).

## Appendix B

### Subproblem: Create split

#### **Component Name: Binary (Subproblem: Create split - numerical)**

##### 1. Concept:

Description: This component divides a numerical attribute in two parts,  $<$  and  $\geq$  from a specific value. The split produced by this component are therefore binary.

Input: Decision table.

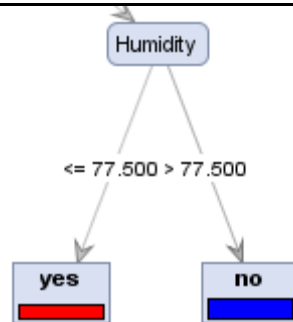
Output: Proposed split.

##### 2. Context:

Application: It represents the easiest way to split numerical attributes. For now only this component is provided for splitting numerical attributes in both approaches.

3. Content: This component can be found in decision tree CART (Breiman 1984) C4.5 (Quinlan 1993).

Example: All records that have value of attribute Humidity  $\leq 77.5$  will be allocated in left branch and others in right branch.



### Component Name: Binary (Subproblem: Create split - categorical)

#### 1. Concept:

Description: This component groups categories of a categorical attribute in two parts. All possible combinations of rearranging categories in two parts can be produced by this component.

Input: Decision table.

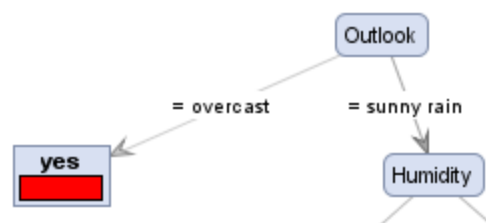
Output: Proposed split.

#### 2. Context:

Application: It represents a computationally demanding way to split categorical attributes. This component, however, can influence producing more accurate splits.

3. Content: This component can be found in decision tree CART (Breiman 1984).

Example: All records that have value of attribute Outlook = "Overcast" will be allocated in left branch and others that have values Outlook = "Sunny, Rain" in right branch



### Component Name: Multiway (Subproblem: Create split - categorical)

#### 1. Concept:

Description: The component produces splits for categorical attributes that have as many leaves as there are categories in an attribute.

Input: Decision table.

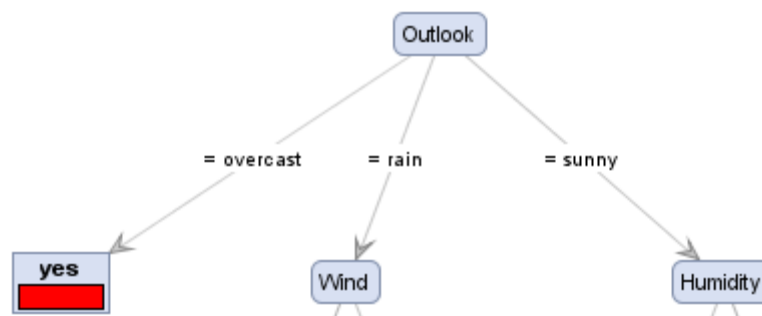
Output: Proposed split.

#### 2. Context:

Application: It represents a computationally effective way of splitting categorical attributes. This component can help discover more interpretable decision trees when there is a few categories in attribute, while maintaining the tree accurate.

3. Content: This component can be found in decision tree C4.5 (Quinlan 1993).

Example: All records that have will be allocated in separate branch for every category of attribute (for attribute Outlook there will be tree branches: overcast, rain and sunny ).



---

### Component Name: Significant (Subproblem: Create split - categorical)

#### 1. Concept:

Description: Groups similar categories into mergers that can produce significant splits.

Input: Decision table with categorical input attributes and categorical output attribute.

Optional:

*Parameter for merging:* Specifies the significance level (alpha) merged categories have to have. The significance level of a merged

category must be greater than 0 and less than or equal to 1. To prevent any merging of categories, specify a value of 1. The default value is 0.05.

*Parameter for splitting:* Specifies the significance level (alpha) for splitting merged categories. The value must be between 0 and 1. The default value is 0.05. Because merging is done hierarchically it can happen that some categories within a merged category are statistically significant with another category in a merged category, but haven't been tested before for significance, because of the hierarchical procedure of merging. It is, therefore, the step of splitting that allows finding near-optimal grouped categories.

Output: Merged categories within input attributes.

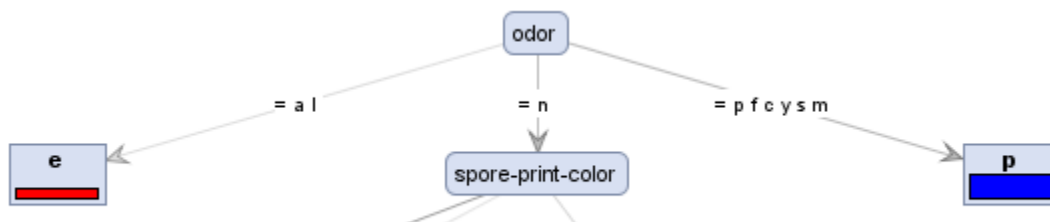
## 2. Context:

Application: Can be used when it is important to join similar categories into a merged categories. For categorial attributes with large number of categories this way of grouping categories can produce more interpretable results.

## 3. Content:

Uses a method for grouping categorical attributes categories into merged categories as described in (Kass 1980).

Example: Attribute Odor has 9 categories, and the records are allocated in 3 branches by the grouped categories: left branch contains records with a,l, central branch with n and right branch with p,f,c,y,s,m.



## Subproblem: Evaluate split

### Component Name: Information Gain (Subproblem: Evaluate Split)

#### 1. Concept:

Description: Evaluates the quality of a split with the Information gain measure. This measure is based on entropy calculation of an input attribute compared to the output attribute. It measures which input attribute describes the output attribute best, and thus reduces entropy.

Input: Split candidate.

Output: Best split.

#### 2. Context:

Application: This measure is computationally demanding and is biased towards choosing attributes with more categories. However, on certain datasets it can produce most accurate results.

3. Content: This split evaluation measure is used in the ID3 algorithm (Quinlan 1986).

### Component Name: Gain ratio (Subproblem: Evaluate Split)

#### 1. Concept:

Description: Evaluates the quality of a split with the Gain ratio measure. This measure is based on entropy calculation of an input attribute compared to the output attribute and takes into account the number of categories in an attribute. It measures which input attribute describes the output attribute best, and thus reduces entropy.

Input: Split candidate.

Output: Best split.

#### 2. Context:

Application: This measure is computationally demanding and is less biased towards choosing attributes with more categories than Information Gain. However, on certain datasets it can produce most accurate results.

3. Content: This split evaluation measure is used in the C4.5 algorithm (Quinlan 1993).



**Component Name: Gini ratio (Subproblem: Evaluate Split)****1. Concept:**

Description: Evaluates the quality of a split with the Gini ratio measure. This measure is based on probability calculation of an input attribute compared to the output attribute. It measures which input attribute describes the output attribute best, and thus reduces impurity of a node. The purest node is chosen as the best split.

Input: Split candidate.

Output: Best split.

**2. Context:**

Application: This measure is not computationally demanding. On certain datasets it can produce most accurate results.

3. Content: This split evaluation measure is used in the CART algorithm (Breiman 1993).

**Component Name: Distance measure (Subproblem: Evaluate Split)****1. Concept:**

Description: Evaluates the quality of a split with the Distance measure. This measure is an improvement of Information gain measure.

Input: Split candidate.

Output: Best split.

**2. Context:**

Application: This measure is computationally demanding and is unbiased towards choosing attributes with more categories. It produces, in general, very accurate results.

3. Content: This split evaluation measure is proposed in (Mantaras 1991) to improve the Information gain measure.

**Component Name: Chi-square test (Subproblem: Evaluate Split)****1. Concept:**

Description: Evaluates the quality of a split with the chi-square test. It checks whether the proposed split is statistically significant.

Input: Split candidate.

Output: Best split.

2. Context:

Application: This measure is biased towards choosing attributes with more categories. It can produce, on some datasets, most accurate results.

3. Content: This split evaluation measure is used in the CHAID algorithm (Kass 1980).

## Subproblem: Stop criteria

### Component Name: Maximum tree depth (Subproblem: Stop criteria)

1. Concept:

Description: Stops growth of decision tree when the maximum tree depth has been reached.

Input: Decision tree in progress, maximum tree depth.

Output: Built decision tree.

2. Context:

Application: This criteria should used when it is important to build trees that shouldn't have more than a specified depth. In some cases, this can prevent overfitting.

3. Content: This stopping criterion is used in almost all decision tree classifiers.

### Component Name: Minimum node size (Subproblem: Stop criteria)

1. Concept:

Description: Stops growth of decision tree on branches when there are not enough cases for a node.

Input: Decision tree in progress, minimum node size.

Output: Built decision tree.

2. Context:

Application: This criteria should used when it is important to build trees that should have nodes with a minimum number of cases. In some cases, this can prevent overfitting.

3. Content: This stop criterion can be used in all decision tree classifiers.

## Subproblem: Prune tree

### **Component Name: Pessimistic error pruning (PEP) (Subproblem: Prune tree)**

#### 1. Concept:

Description: This method uses a pessimistic criterion to decide which subtree to replace with a node.

Input: Decision tree. Confidence  $(0, 0.5]$  If this value is closer to 0.5 more severe pruning is performed.

Output: Pruned decision tree.

#### 2. Context:

Application: Can be used to reduce the tree in order to get more accurate or more understandable trees.

3. Content: This method is proposed in (Quinlan 1993).

### **Component Name: Minimum leaf size (Subproblem: Stop criteria)**

#### 1. Concept:

Description: Stops growth of decision tree on branches when there are not enough cases for a leaf.

Input: Decision tree in progress, minimum leaf size.

Output: Built decision tree.

#### 2. Context:

Application: This criteria should be used when it is important to build trees that should have leaves with a minimum number of cases. In some cases, this can prevent overfitting.

3. Content: This pruning criterion can be used in all decision tree classifiers.

## References

1. Asuncion A, Newman DJ (2007) UCI Machine Learning Repository, University of California, School of Information and Computer Science. [[www.ics.uci.edu/~mlearn/MLRepository.html](http://www.ics.uci.edu/~mlearn/MLRepository.html)].
2. Breiman L, Friedman J, Stone CJ, Olshen RA (1984) Classification and Regression Trees, CRC Press
3. Kass GV (1980) An Exploratory Technique for Investigating Large Quantities of Categorical Data, Applied Statistics, 29 (2), p 119-127
4. Loh WY, Shih YS (1997) Split selection methods for classification trees, Statistica Sinica 7, p. 815-840
5. Mantaras RL (1991) A Distance-Based Attribute Selection Measure for Decision Tree Induction, Machine Learning
6. Mierswa I, Wurst M, Klinkenberg R, Scholz M, Euler T (2006) YALE: Rapid Prototyping for Complex Data Mining Tasks. In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM Press
7. Quinlan JR (1986) Induction of Decision Trees. Machine Learning 1, p. 81-106
8. Quinlan JR (1993) C4.5 Programs for Machine Learning, Morgan Kaufmann
9. Sonnenburg S, Braun ML, Ong CS, Bengio S, Bottou L, Holmes G, LeCun Y, Mueller KR, Pereira F, Rasmussen CE, Raetsch G, Schoelkopf B, Smola A (2007) The Need for Open Source Software in Machine Learning, Journal of Machine Learning Research 8, p. 2443-2466
10. Tracz W (1990). Where does reuse start?. ACM SIGSOFT Software Engineering Notes 15:42-46
11. Wu X, Kumar V, Quinlan JR, Ghosh J, Yang Q, Motoda H, McLachlan GJ, Ng A, Liu B, Yu PS, Zhou ZH, Steinbach M, Hand DJ, Steinberg D (2008) Top 10 algorithms in data mining, Knowledge information systems 14, p 1-37