Orange Multitarget v0.9.3 documentation

Release 0.9.3

Bioinformatics Laboratory, FRI UL

October 23, 2013

CONTENTS

Orange Multitarget is an add-on for Orange data mining software package. It extends Orange by providing methods that allow for classification of datasets with multiple classes.

CHAPTER

ONE

SCRIPTING REFERENCE

1.1 Multi-target prediction (`multitarget`)

1.1.1 Examples

```
import Orange

data = Orange.data.Table('multitarget:bridges.tab')

cl1 = Orange.multitarget.binary.BinaryRelevanceLearner( \
        learner = Orange.classification.majority.MajorityLearner, name="Binary - Maj")
cl2 = Orange.multitarget.binary.BinaryRelevanceLearner( \
        learner = Orange.classification.tree.SimpleTreeLearner, name="Binary - Tree")

learners = [cl1,cl2]

results = Orange.evaluation.testing.cross_validation(learners, data)

print "Classification - bridges.tab"
print "%18s  %6s  %8s  %8s" % ("Learner    ", "LogLoss", "Mean Acc", "Glob Acc")
for i in range(len(learners)):
    print "%18s  %1.4f    %1.4f    %1.4f" % (learners[i].name,
    Orange.multitarget.scoring.mt_average_score(results, Orange.evaluation.scoring.logloss)[i],
    Orange.multitarget.scoring.mt_mean_accuracy(results)[i],
    Orange.multitarget.scoring.mt_global_accuracy(results)[i])
```

1.1.2 Examples

```
import Orange

data = Orange.data.Table('multitarget:bridges.tab')

cl1 = Orange.multitarget.chain.ClassifierChainLearner( \
        learner = Orange.classification.majority.MajorityLearner, name="CChain - Maj")
cl2 = Orange.multitarget.chain.ClassifierChainLearner( \
        learner = Orange.classification.tree.SimpleTreeLearner, name="CChain - Tree")
cl3 = Orange.multitarget.chain.EnsembleClassifierChainLearner( \
        learner = Orange.classification.tree.SimpleTreeLearner, n_chains=50, sample_size=0.25, name="

learners = [cl1,cl2,cl3]
```

```
results = Orange.evaluation.testing.cross_validation(learners, data)

print "Classification - bridges.tab"
print "%18s  %6s  %8s  %8s" % ("Learner    ", "LogLoss", "Mean Acc", "Glob Acc")
for i in range(len(learners)):
    print "%18s  %1.4f    %1.4f    %1.4f" % (learners[i].name,
    Orange.multitarget.scoring.mt_average_score(results, Orange.evaluation.scoring.logloss)[i],
    Orange.multitarget.scoring.mt_mean_accuracy(results)[i],
    Orange.multitarget.scoring.mt_global_accuracy(results)[i])
```

1.1.3 Neural Network Learner (`neural`)

class Orange.classification.neural.**NeuralNetworkLearner**(*name='NeuralNetwork'*, *n_mid=10*, *reg_fact=1*, *max_iter=300*, *normalize=True*, *rand=None*)

Bases: Orange.classification.Learner

NeuralNetworkLearner implements a multilayer perceptron. Learning is performed by minimizing an L2-regularized cost function with scipy's implementation of L-BFGS. The current implementations is limited to a single hidden layer.

Regression is currently not supported.

Parameters

- **name** (*string*) – learner name.
- **n_mid** (*int*) – Number of nodes in the hidden layer
- **reg_fact** (*float*) – Regularization factor.
- **max_iter** (*int*) – Maximum number of iterations.
- **normalize** (*bool*) – Normalize the data prior to learning (subtract each column by the mean and divide by the standard deviation)

Return type Orange.multitarget.neural.neuralNetworkLearner or Orange.multitarget.chain.NeuralNetworkClassifier

class Orange.classification.neural.**NeuralNetworkClassifier**(***kwargs*)

Uses the classifier induced by the NeuralNetworkLearner.

Parameters **name** (*string*) – name of the classifier.

Example of multi-target usage:

```
import Orange

l1 = Orange.multitarget.neural.NeuralNetworkLearner(n_mid=15, reg_fact=0.1, max_iter=100, name="Neura
l2 = Orange.multitarget.binary.BinaryRelevanceLearner(
        learner = Orange.classification.majority.MajorityLearner, name = "Majority")
learners = [l1, l2]

data = Orange.data.Table('multitarget:flare.tab')

results = Orange.evaluation.testing.cross_validation(learners, data, 3)

print "Classification - flare.tab"
print "%18s  %6s  %8s  %8s" % ("Learner    ", "LogLoss", "Mean Acc", "Glob Acc")
for i in range(len(learners)):
```

```
    print "%18s  %1.4f    %1.4f    %1.4f" % (learners[i].name,
    Orange.multitarget.scoring.mt_average_score(results, Orange.evaluation.scoring.logloss)[i],
    Orange.multitarget.scoring.mt_mean_accuracy(results)[i],
    Orange.multitarget.scoring.mt_global_accuracy(results)[i])


# Neural Networks do not work with missing values, the missing values need to be imputed
data = Orange.data.Table('multitarget:bridges.tab')
imputer = Orange.feature.imputation.AverageConstructor()
imputer = imputer(data)
imp_data = imputer(data)

results = Orange.evaluation.testing.cross_validation(learners, imp_data, 3)

print "Classification - imputed bridges.tab"
print "%18s  %6s  %8s  %8s" % ("Learner    ", "LogLoss", "Mean Acc", "Glob Acc")
for i in range(len(learners)):
    print "%18s  %1.4f    %1.4f    %1.4f" % (learners[i].name,
    Orange.multitarget.scoring.mt_average_score(results, Orange.evaluation.scoring.logloss)[i],
    Orange.multitarget.scoring.mt_mean_accuracy(results)[i],
    Orange.multitarget.scoring.mt_global_accuracy(results)[i])
```

1.1.4 PLS Regression Learner

Partial least sqaures regression (`PLS`)

Partial least squares regression is a statistical method for simultaneous prediction of multiple response variables. Orange's implementation is based on Scikit learn python implementation.

The following code shows how to fit a PLS regression model on a multi-target data set.

```
import Orange
data = Orange.data.Table("multitarget-synthetic.tab")
learner = Orange.multitarget.pls.PLSRegressionLearner()
classifier = learner(data)
```

class `Orange.regression.pls.`**`PLSRegressionLearner`**(**args*, ***kwargs*)

Fit the partial least squares regression model, i.e. learn the regression parameters. The implementation is based on Scikit learn python implementation

The class is derived from `Orange.regression.base.BaseRegressionLearner` that is used for preprocessing the data (continuization and imputation) before fitting the regression parameters

`deflationMode`

A deprecated member 'deflationMode'. Use 'deflation_mode' instead.

`fit`(*X*, *Y*)

Fit all unknown parameters, i.e. weights, scores, loadings (for x and y) and regression coefficients. Return a dict with all of the parameters.

`maxIter`

A deprecated member 'maxIter'. Use 'max_iter' instead.

`nComp`

A deprecated member 'nComp'. Use 'n_comp' instead.

class `Orange.regression.pls.`**`PLSRegression`**(**args*, ***kwargs*)

Predict values of the response variables based on the values of independent variables.

Basic notations: n - number of data instances p - number of independent variables q - number of reponse variables

T
A n x n_comp numpy array of x-scores

U
A n x n_comp numpy array of y-scores

W
A p x n_comp numpy array of x-weights

C
A q x n_comp numpy array of y-weights

P
A p x n_comp numpy array of x-loadings

Q
A q x n_comp numpy array of y-loading

coefs
A p x q numpy array coefficients of the linear model: Y = X coefs + E

x_vars
Predictor variables

y_vars
Response variables

muX
A deprecated member 'muX'. Use 'mu_x' instead.

muY
A deprecated member 'muY'. Use 'mu_y' instead.

sigmaX
A deprecated member 'sigmaX'. Use 'sigma_x' instead.

sigmaY
A deprecated member 'sigmaY'. Use 'sigma_y' instead.

to_string()
Pretty-prints the coefficient of the PLS regression model.

xVars
A deprecated member 'xVars'. Use 'x_vars' instead.

yVars
A deprecated member 'yVars'. Use 'y_vars' instead.

Utility functions

Orange.regression.pls.**normalize_matrix**(*X*)
Normalize a matrix column-wise: subtract the means and divide by standard deviations. Returns the standardized matrix, sample mean and standard deviation

Parameters **X** (numpy.array) – data matrix

Orange.regression.pls.**nipals_xy**(**args*, ***kwargs*)
NIPALS algorithm; returns the first left and rigth singular vectors of X'Y.

Parameters

- **Y** (*X,*) – data matrix
- **mode** (*string*) – possible values “PLS” (default) or “CCA”
- **max_iter** (*int*) – maximal number of iterations (default: 500)
- **tol** (*a not negative float*) – tolerance parameter; if norm of difference between two successive left singular vectors is less than tol, iteration is stopped

Orange.regression.pls.**svd_xy**(*X*, *Y*)

Return the first left and right singular vectors of X’Y.

Parameters **Y** (*X,*) – data matrix

Examples The following code predicts the values of output variables for the first two instances in `data`.

```
print "Prediction for the first 2 data instances: \n"
for d in data[:2]:
    print "Actual    ", d.get_classes()
    print "Predicted ", classifier(d)
    print
```

```
Actual     [<orange.Value 'Y1'='0.490'>, <orange.Value 'Y2'='1.237'>, <orange.Value 'Y3'='1.808'>, <o
Predicted  [<orange.Value 'Y1'='0.613'>, <orange.Value 'Y2'='0.826'>, <orange.Value 'Y3'='1.084'>, <o

Actual     [<orange.Value 'Y1'='0.167'>, <orange.Value 'Y2'='-0.664'>, <orange.Value 'Y3'='-1.378'>,
Predicted  [<orange.Value 'Y1'='0.058'>, <orange.Value 'Y2'='-0.706'>, <orange.Value 'Y3'='-1.420'>,
```

To see the coefficient of the model, print the model:

```
print 'Regression coefficients:\n', classifier
```

```
Regression coefficients:
                   Y1           Y2           Y3           Y4
      X1        0.714        2.153        3.590       -0.078
      X2       -0.238       -2.500       -4.797       -0.036
      X3        0.230       -0.314       -0.880       -0.060
```

Note that coefficients are stored in a matrix since the model predicts values of multiple outputs.

1.1.5 Examples

```
import Orange

l1 = Orange.multitarget.pls.PLSClassificationLearner(n_mid=15, reg_fact=0.1, max_iter=100, name="PLS"
l2 = Orange.multitarget.binary.BinaryRelevanceLearner(
        learner = Orange.classification.majority.MajorityLearner, name = "Majority")
learners = [l1, l2]

data = Orange.data.Table('multitarget:flare.tab')

results = Orange.evaluation.testing.cross_validation(learners, data, 3)

print "Classification - flare.tab"
print "%18s  %6s  %8s  %8s" % ("Learner    ", "LogLoss", "Mean Acc", "Glob Acc")
for i in range(len(learners)):
    print "%18s  %1.4f    %1.4f    %1.4f" % (learners[i].name,
```

```
    Orange.multitarget.scoring.mt_average_score(results, Orange.evaluation.scoring.logloss)[i],
    Orange.multitarget.scoring.mt_mean_accuracy(results)[i],
    Orange.multitarget.scoring.mt_global_accuracy(results)[i])

# REGRESSION
l1 = Orange.multitarget.pls.PLSRegressionLearner(name="PLS")
l2 = Orange.multitarget.binary.BinaryRelevanceLearner(
        learner = Orange.regression.mean.MeanLearner, name = "Majority")
learners = [l1, l2]
# PLSClassifier do not work with missing values, the missing values need to be imputed
data = Orange.data.Table('multitarget-synthetic')

results = Orange.evaluation.testing.cross_validation(learners, data, 3)

print "Regression - multitarget-synthetic.tab"
print "%18s  %6s" % ("Learner    ", "RMSE")
for i in range(len(learners)):
    print "%18s  %1.4f" % (learners[i].name,
    Orange.multitarget.scoring.mt_average_score(results, Orange.evaluation.scoring.RMSE)[i])
```

1.1.6 Examples

```
import Orange

data = Orange.data.Table('multitarget:bridges.tab')

cl1 = Orange.multitarget.binary.BinaryRelevanceLearner( \
    learner = Orange.classification.majority.MajorityLearner, name="Majority")
cl2 = Orange.multitarget.tree.ClusteringTreeLearner(name="CTree")

learners = [cl1,cl2]

results = Orange.evaluation.testing.cross_validation(learners, data)

print "%18s  %7s    %6s  %10s   %8s  %8s" % \
("Learner    ", "LogLoss", "Brier", "Inf. Score", "Mean Acc", "Glob Acc")
for i in range(len(learners)):
    print "%18s   %1.4f    %1.4f     %+2.4f     %1.4f    %1.4f" % (learners[i].name,

    # Calculate average logloss
    Orange.multitarget.scoring.mt_average_score(results, \
        Orange.evaluation.scoring.logloss)[i],
    # Calculate average Brier score
    Orange.multitarget.scoring.mt_average_score(results, \
        Orange.evaluation.scoring.Brier_score)[i],
    # Calculate average Information Score
    Orange.multitarget.scoring.mt_average_score(results, \
        Orange.evaluation.scoring.IS)[i],
    # Calculate mean accuracy
    Orange.multitarget.scoring.mt_mean_accuracy(results)[i],
    # Calculate global accuracy
    Orange.multitarget.scoring.mt_global_accuracy(results)[i])
```

Multi-target prediction tries to achieve better prediction accuracy or speed through prediction of multiple dependent variables at once. It works on multi-target data, which is also supported by Orange's tab file format using multiclass directive.

List of supported learners:

- *<no title>*
- *Examples*
- *Examples*
- *Neural Network Learner (neural)*
- *PLS Regression Learner*

Additionally `orangecontrib.earth.EarthLearner` from the orangecontrib.earth package is also supports multi-target predictions.

For evaluation of multi-target methods, see the corresponding section in *Examples*.

The addon also includes three sample datasets:

- **bridges.tab** - dataset with 5 multi-class class variables
- **flare.tab** - dataset with 3 multi-class class variables
- **emotions.tab** - dataset with 6 binary class variables (a multi-label dataset)

Example of loading an included dataset:

```
import Orange
data = Orange.data.Table('multitarget:bridges.tab')
```

CHAPTER

TWO

WIDGETS

2.1 Binary Relevance

2.1.1 Signals

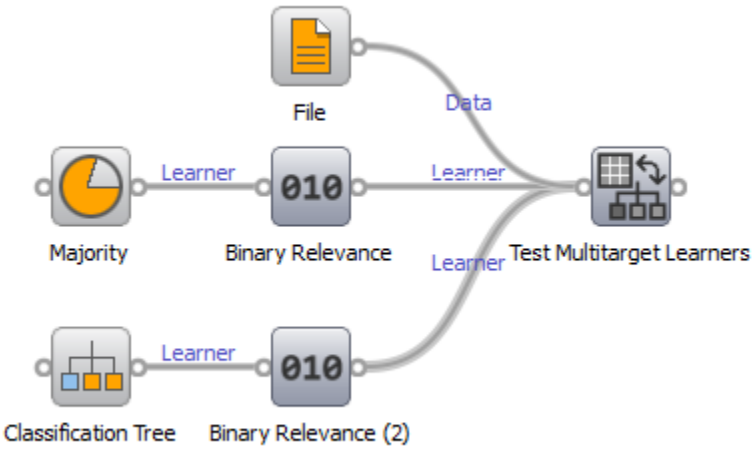
Inputs:

- **Learner** The base learner used in the ensemble technique.

Outputs:

- Learner or Classifier

2.1.2 Description



Binary relevance learner takes a single-target learner and with it creates a classifier for every class variable in the data.

2.2 Classifier Chain

2.2.1 Signals

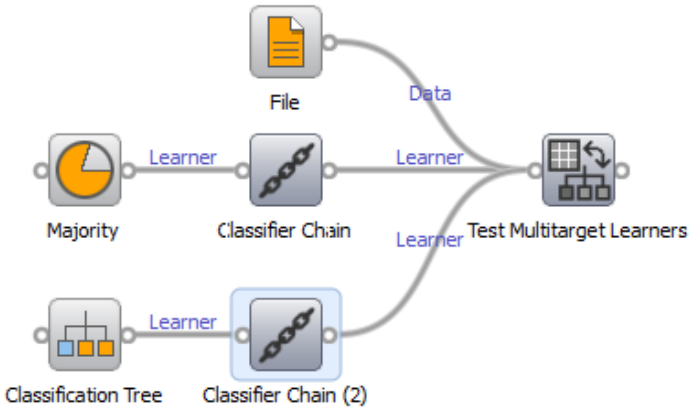
Inputs:

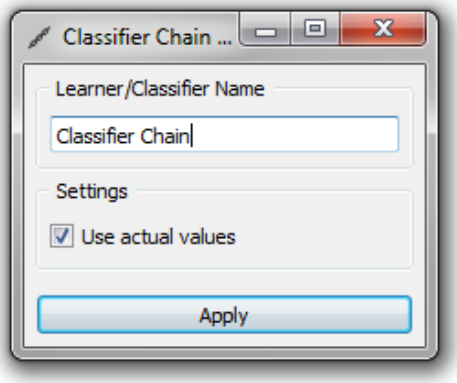
- **Learner** The base learner used in the ensemble technique.

Outputs:

- Learner or Classifier

2.2.2 Description





Classifier chain learner takes a single-target learner and with it creates a classifier for every class variable in the data. Every time a classifier is created, the values of that class variable are added to features. The order in which the class variables are chosen is random.

2.2.3 Setting:

- **Use actual values** If checked, the values added into features are actual values from the data. Otherwise the values predicted by the classifier are used.

2.3 Ensemble Classifier Chain

2.3.1 Signals

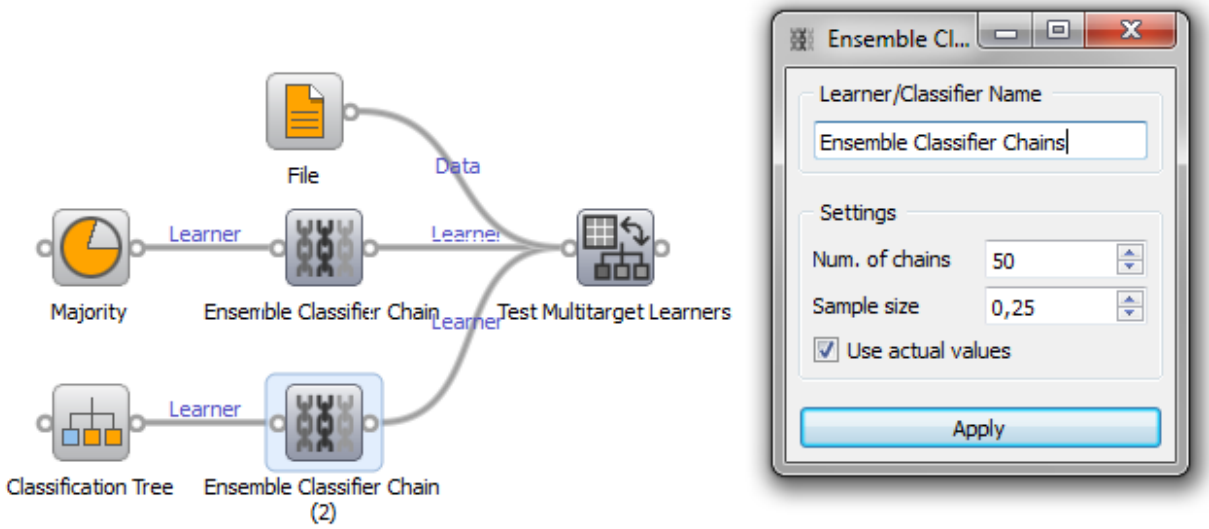
Inputs:

- **Learner** The base learner used in the ensemble technique.

Outputs:

- Learner or Classifier

2.3.2 Description



Ensemble classifier chain learner takes a single-target learner and with it creates a number of classifier chains. Each chain is constructed on a random sample of the dataset.

2.3.3 Setting:

- **Number of chains** Number of classifier chains that are built.
- **Sample size** The size of the random sample taken from the dataset for each chain.
- **Use actual values** If checked, the values added into features are actual values from the data. Otherwise the values predicted by the classifier are used.

2.4 PLS Classification

PLS

2.4.1 Signals

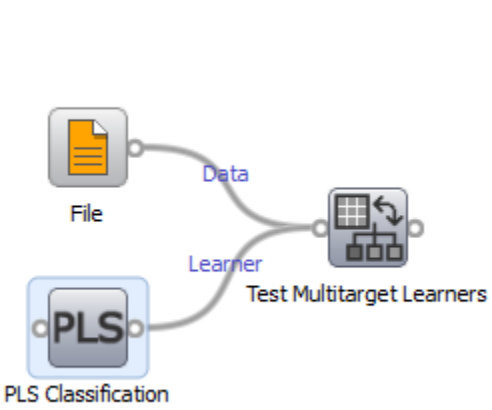
Inputs:

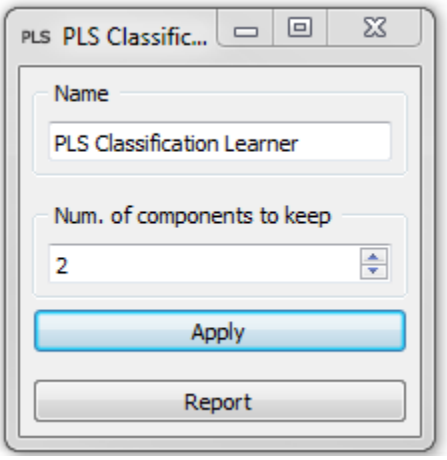
- **Data** Data to be used for learning.

Outputs:

- Learner or Classifier

2.4.2 Description





PLS is originally a regression technique. PLSClassification wraps Orange's implementation of PLS into a classifier. Usage is equall to all learners, settings are described below.

2.4.3 Settings

- Num. of components to keep The number of components in the matrix that PLS constructs for regression and classification.

2.5 Clustering Tree

2.5.1 Signals

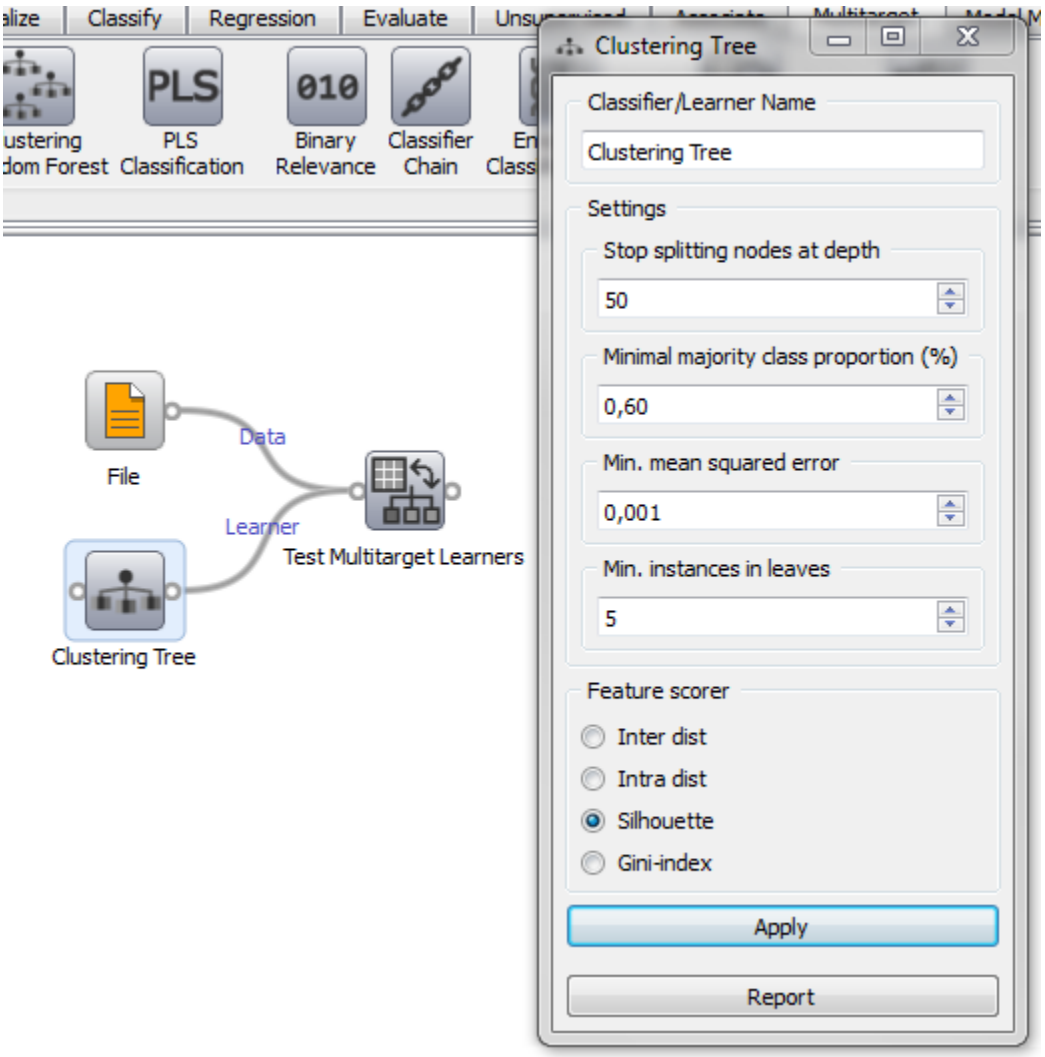
Inputs:

- **Data** Data to be used for learning.

Outputs:

- Learner or Classifier

2.5.2 Description



Clustering trees are similiar to classic decision trees, to select features they measure the distance between clusters the featers would create by splitting the dataset. Usage is simple, the settings are described below.

2.5.3 Settings

- Stop splitting nodes at depth

  Maximal depth of tree.

- Minimal majority class proportion

  Minimal proportion of the majority class value each of the class variables has to reach to stop induction (only used for classification).

- Min mean squared error

  Minimal mean squared error each of the class variables has to reach to stop induction (only used for regression).

- Min. instances in leaves

Minimal number of instances in leaves. Instance count is weighed.

- Feature scorer
  - Inter dist (default) - Euclidean distance between centroids of clusters
  - Intra dist - average Euclidean distance of each member of a cluster to the centroid of that cluster
  - Silhouette - silhouette (http://en.wikipedia.org/wiki/Silhouette_(clustering)) measure calculated with euclidean distances between clusters instead of elements of a cluster.
  - Gini-index - calculates the Gini-gain index, should be used with class variables with nominal values

2.6 Clustering Random Forest

2.6.1 Signals

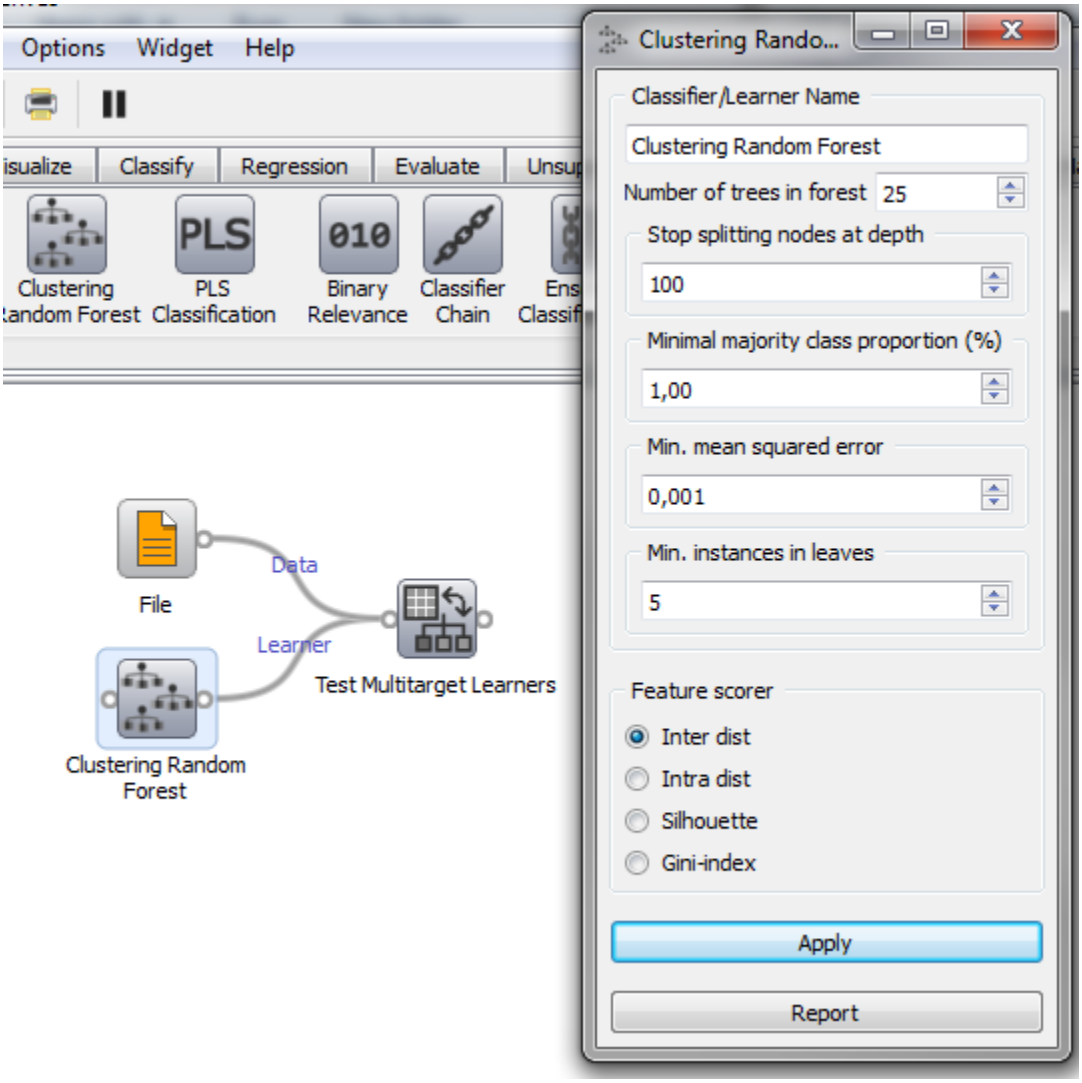
Inputs:

- **Data** Data to be used for learning.

Outputs:

- Learner or Classifier

2.6.2 Description



A clustering random forest is a random forest consisting of clustering trees. The usage is straightforward and the setting are described below.

2.6.3 Settings

- Number of trees in forest

  Number of trees in forest.

- Stop splitting nodes at depth

  Maximal depth of tree.

- Minimal majority class proportion

  Minimal proportion of the majority class value each of the class variables has to reach to stop induction (only used for classification).

- Min mean squared error

Minimal mean squared error each of the class variables has to reach to stop induction (only used for regression).

- Min. instances in leaves

  Minimal number of instances in leaves. Instance count is weighed.

- Feature scorer
  - Inter dist (default) - Euclidean distance between centroids of clusters
  - Intra dist - average Euclidean distance of each member of a cluster to the centroid of that cluster
  - Silhouette - silhouette (http://en.wikipedia.org/wiki/Silhouette_(clustering)) measure calculated with euclidean distances between clusters instead of elements of a cluster.
  - Gini-index - calculates the Gini-gain index, should be used with class variables with nominal values

2.7 Neural Network

2.7.1 Signals

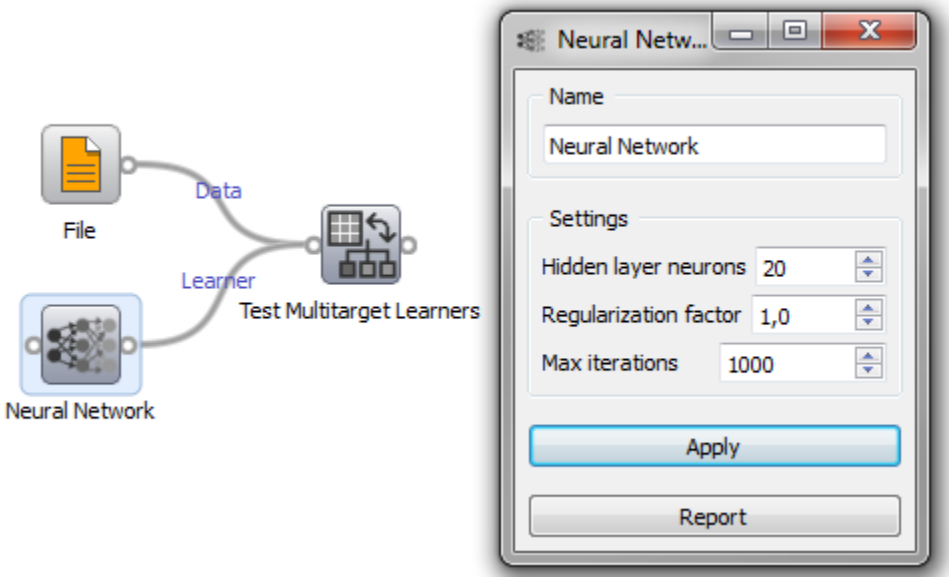
Inputs:

- **Data** Data to be used for learning.

Outputs:

- Learner or Classifier

2.7.2 Description



Neural networks are a complex technique accessible in a simple widget. Setting are described below.

2.7.3 Settings

- Hidden layer neurons

  The number of neurons in the hidden layer.

- Reguralization factor

  Regularization factor controls overfitting.

- Max iterations

  Maximal number of iterations the optimization algorithm can make.

2.8 Test Multitarget Learners

2.8.1 Signals

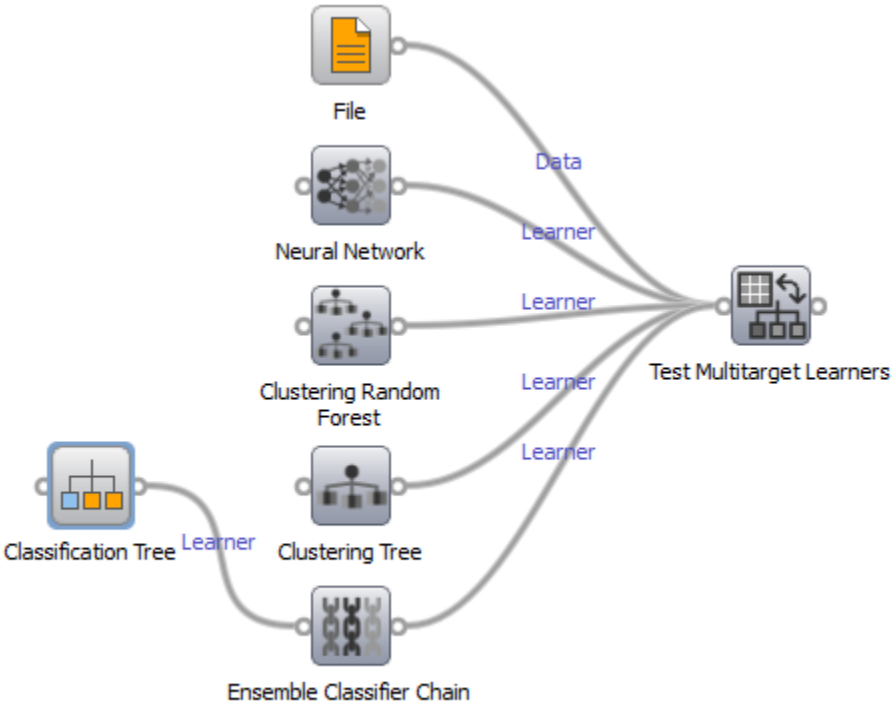
Inputs:

- **Data** Data to be used for testing.
- Seoerate Test Data Separate data for testing
- Learner One or more learning algorithms

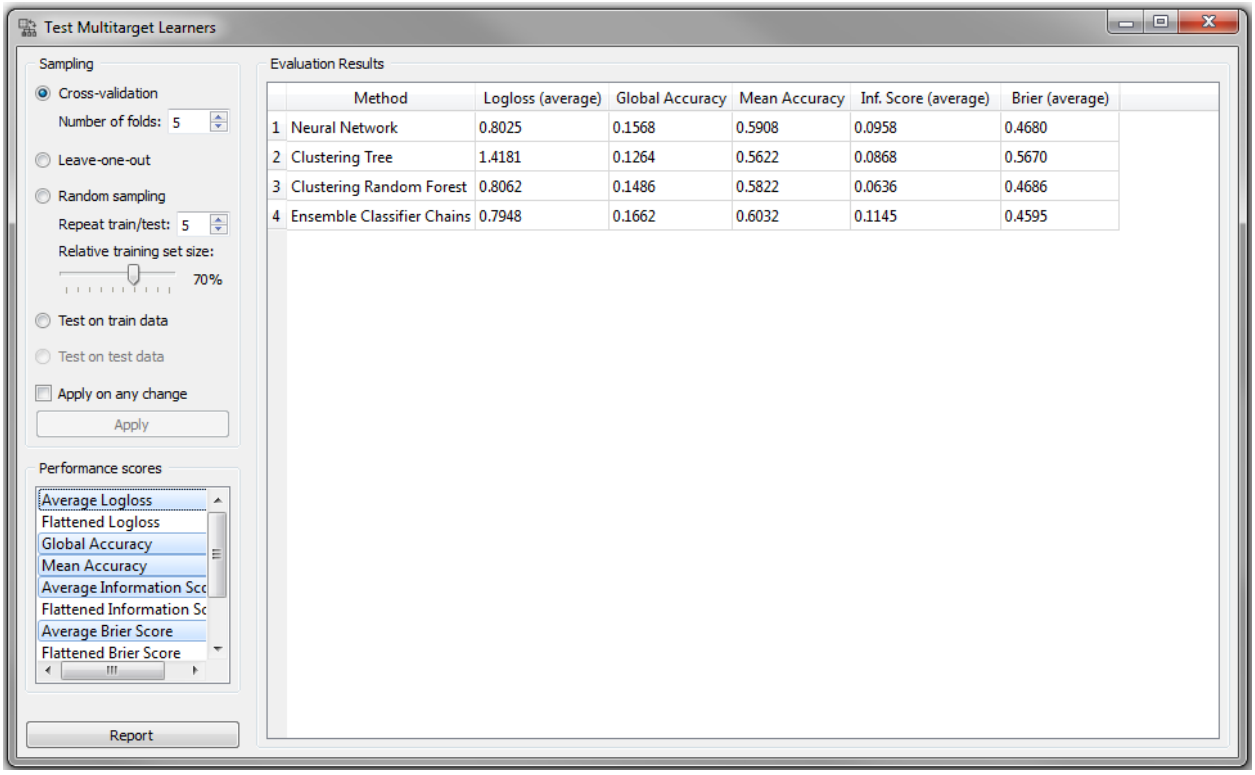
Outputs:

- **Evaluation results** Results of testing the algorithms

2.8.2 Description



This widget is used for testing built learners. We provide the data and a number of learners we want to compare to the inputs of this wiget. Inside the widget we then select the method of testing and the scores we wish to measure. Results are displayed on the table to the right.



2.8.3 Settings

- Sampling

  Here we can choose the method of sampling for testing the learners. Available methods are cross-validation, leave-one-out testing, random sampling, test on train data and test on test data (this requires additional test data on input)

- Performance Scorers

  A list of scorers is available, by clicking on one of them we either add or remove a scorer from the table of results.

CHAPTER

THREE

INSTALLATION

To install Multitarget add-on for Orange from PyPi run:

```
pip install Orange-Multitarget
```

To install it from source code run:

```
python setup.py install
```

To build Python egg run:

```
python setup.py bdist_egg
```

To install add-on in development mode run:

```
python setup.py develop
```

CHAPTER

FOUR

SOURCE CODE AND ISSUE TRACKER

Source code is available on Bitbucket. For issues and wiki we use Trac.

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

O