Orange Data Mining Library Documentation

Release 3

Orange Data Mining

Sep 15, 2017

Contents

1 Tutorial **1**

1.1 The Data . . . 1
1.1.1 Data Input . . . 1
1.1.2 Saving the Data . . . 3
1.1.3 Exploration of the Data Domain . . . 3
1.1.4 Data Instances . . . 4
1.1.5 Orange Data Sets and NumPy . . . 5
1.1.6 Meta Attributes . . . 6
1.1.7 Missing Values . . . 7
1.1.8 Data Selection and Sampling . . . 8
1.2 Classification . . . 8
1.2.1 Learners and Classifiers . . . 9
1.2.2 Probabilistic Classification . . . 10
1.2.3 Cross-Validation . . . 10
1.2.4 Handful of Classifiers . . . 10
1.3 Regression . . . 12
1.3.1 Handful of Regressors . . . 12
1.3.2 Cross Validation . . . 13

2 Reference **15**

2.1 Data model (`data`) . . . 15
2.1.1 Data Storage (`storage`) . . . 16
2.1.2 Data Table (`table`) . . . 18
2.1.3 SQL table (`data.sql`) . . . 22
2.1.4 Domain description (`domain`) . . . 23
2.1.5 Variable Descriptors (`variable`) . . . 27
2.1.6 Values (`value`) . . . 32
2.1.7 Data Instance (`instance`) . . . 33
2.1.8 Data Filters (`filter`) . . . 35
2.1.9 Loading and saving data (`io`) . . . 37
2.2 Data Preprocessing (`preprocess`) . . . 40
2.2.1 Impute . . . 40
2.2.2 Discretization . . . 40
2.2.3 Continuization . . . 41
2.2.4 Normalization . . . 44
2.2.5 Randomization . . . 44

2.2.6 Remove . . . 45
2.2.7 Feature selection . . . 45
2.2.8 Preprocessors . . . 49
2.3 Classification (classification) . . . 49
2.3.1 Logistic Regression . . . 49
2.3.2 Random Forest . . . 49
2.3.3 Simple Random Forest . . . 50
2.3.4 Softmax Regression . . . 50
2.3.5 k-Nearest Neighbors . . . 51
2.3.6 Naive Bayes . . . 51
2.3.7 Support Vector Machines . . . 52
2.3.8 Linear Support Vector Machines . . . 52
2.3.9 Nu-Support Vector Machines . . . 52
2.3.10 One Class Support Vector Machines . . . 53
2.3.11 Classification Tree . . . 53
2.3.12 Simple Tree . . . 54
2.3.13 Majority Classifier . . . 54
2.3.14 Elliptic Envelope . . . 54
2.3.15 Neural Network . . . 55
2.3.16 CN2 Rule Induction . . . 55
2.4 Regression (regression) . . . 57
2.4.1 Linear Regression . . . 57
2.4.2 Polynomial . . . 58
2.4.3 Mean . . . 58
2.4.4 Random Forest . . . 59
2.4.5 Simple Random Forest . . . 59
2.4.6 Regression Tree . . . 60
2.4.7 Neural Network . . . 61
2.5 Clustering (clustering) . . . 61
2.5.1 Hierarchical (hierarchical) . . . 61
2.6 Distance (distance) . . . 62
2.6.1 Handling discrete and missing data . . . 64
2.6.2 Supported distances . . . 64
2.7 Evaluation (evaluation) . . . 66
2.7.1 Sampling procedures for testing models (testing) . . . 66
2.7.2 Scoring methods (scoring) . . . 68
2.8 Projection (projection) . . . 71
2.8.1 PCA . . . 71
2.9 Miscellaneous (misc) . . . 73
2.9.1 Distance Matrix (distmatrix) . . . 73

Bibliography **75**

Python Module Index **77**

CHAPTER 1

Tutorial

This is a gentle introduction on scripting in Orange , a Python 3 data mining library. We here assume you have already downloaded and installed Orange from its github repository and have a working version of Python. In the command line or any Python environment, try to import Orange. Below, we used a Python shell:

```
% python
>>> import Orange
>>> Orange.version.version
'3.2.dev0+8907f35'
>>>
```

If this leaves no error and warning, Orange and Python are properly installed and you are ready to continue with the Tutorial.

The Data

This section describes how to load the data in Orange. We also show how to explore the data, perform some basic statistics, and how to sample the data.

Data Input

Orange can read files in native tab-delimited format, or can load data from any of the major standard spreadsheet file type, like CSV and Excel. Native format starts with a header row with feature (column) names. Second header row gives the attribute type, which can be continuous, discrete, time, or string. The third header line contains meta information to identify dependent features (class), irrelevant features (ignore) or meta features (meta). More detailed specification is available in *Loading and saving data (io)*. Here are the first few lines from a data set `lenses.tab`:

```
age       prescription  astigmatic    tear_rate     lenses
discrete  discrete      discrete      discrete      discrete
                                                    class
young     myope         no            reduced       none
young     myope         no            normal        soft
```

```
young      myope          yes            reduced        none
young      myope          yes            normal         hard
young      hypermetrope   no             reduced        none
```

Values are tab-limited. This data set has four attributes (age of the patient, spectacle prescription, notion on astigmatism, and information on tear production rate) and an associated three-valued dependent variable encoding lens prescription for the patient (hard contact lenses, soft contact lenses, no lenses). Feature descriptions could use one letter only, so the header of this data set could also read:

```
age        prescription   astigmatic     tear_rate      lenses
d          d              d              d              d
                                                        c
```

The rest of the table gives the data. Note that there are 5 instances in our table above. For the full data set, check out or download `lenses.tab`) to a target directory. You can also skip this step as Orange comes preloaded with several demo data sets, lenses being one of them. Now, open a python shell, import Orange and load the data:

```
>>> import Orange
>>> data = Orange.data.Table("lenses")
>>>
```

Note that for the file name no suffix is needed; as Orange checks if any files in the current directory are of the readable type. The call to `Orange.data.Table` creates an object called `data` that holds your data set and information about the lenses domain:

```
>>> data.domain.attributes
(DiscreteVariable('age', values=['pre-presbyopic', 'presbyopic', 'young']),
 DiscreteVariable('prescription', values=['hypermetrope', 'myope']),
 DiscreteVariable('astigmatic', values=['no', 'yes']),
 DiscreteVariable('tear_rate', values=['normal', 'reduced']))
>>> data.domain.class_var
DiscreteVariable('lenses', values=['hard', 'none', 'soft'])
>>> for d in data[:3]:
   ...:     print(d)
   ...:
[young, myope, no, reduced | none]
[young, myope, no, normal | soft]
[young, myope, yes, reduced | none]
>>>
```

The following script wraps-up everything we have done so far and lists first 5 data instances with `soft` perscription:

```
import Orange
data = Orange.data.Table("lenses")
print("Attributes:", ", ".join(x.name for x in data.domain.attributes))
print("Class:", data.domain.class_var.name)
print("Data instances", len(data))

target = "soft"
print("Data instances with %s prescriptions:" % target)
atts = data.domain.attributes
for d in data:
    if d.get_class() == target:
        print(" ".join(["%14s" % str(d[a]) for a in atts]))
```

Note that data is an object that holds both the data and information on the domain. We show above how to access attribute and class names, but there is much more information there, including that on feature type, set of values for

categorical features, and other.

Saving the Data

Data objects can be saved to a file:

```
>>> data.save("new_data.tab")
>>>
```

This time, we have to provide the file extension to specify the output format. An extension for native Orange's data format is ".tab". The following code saves only the data items with myope perscription:

```
import Orange
data = Orange.data.Table("lenses")
myope_subset = [d for d in data if d["prescription"] == "myope"]
new_data = Orange.data.Table(data.domain, myope_subset)
new_data.save("lenses-subset.tab")
```

We have created a new data table by passing the information on the structure of the data (`data.domain`) and a subset of data instances.

Exploration of the Data Domain

Data table stores information on data instances as well as on data domain. Domain holds the names of attributes, optional classes, their types and, and if categorical, the value names. The following code:

```
import Orange

data = Orange.data.Table("imports-85.tab")
n = len(data.domain.attributes)
n_cont = sum(1 for a in data.domain.attributes if a.is_continuous)
n_disc = sum(1 for a in data.domain.attributes if a.is_discrete)
print("%d attributes: %d continuous, %d discrete" % (n, n_cont, n_disc))

print("First three attributes:",
      ", ".join(data.domain.attributes[i].name for i in range(3)))

print("Class:", data.domain.class_var.name)
```

outputs:

```
25 attributes: 14 continuous, 11 discrete
First three attributes: symboling, normalized-losses, make
Class: price
```

Orange's objects often behave like Python lists and dictionaries, and can be indexed or accessed through feature names:

```
print("First attribute:", data.domain[0].name)
name = "fuel-type"
print("Values of attribute '%s': %s" %
      (name, ", ".join(data.domain[name].values)))
```

The output of the above code is:

```
First attribute: symboling
Values of attribute 'fuel-type': diesel, gas
```

Data Instances

Data table stores data instances (or examples). These can be index or traversed as any Python list. Data instances can be considered as vectors, accessed through element index, or through feature name.

```
import Orange

data = Orange.data.Table("iris")
print("First three data instances:")
for d in data[:3]:
    print(d)

print("25-th data instance:")
print(data[24])

name = "sepal width"
print("Value of '%s' for the first instance:" % name, data[0][name])
print("The 3rd value of the 25th data instance:", data[24][2])
```

The script above displays the following output:

```
First three data instances:
[5.100, 3.500, 1.400, 0.200 | Iris-setosa]
[4.900, 3.000, 1.400, 0.200 | Iris-setosa]
[4.700, 3.200, 1.300, 0.200 | Iris-setosa]
25-th data instance:
[4.800, 3.400, 1.900, 0.200 | Iris-setosa]
Value of 'sepal width' for the first instance: 3.500
The 3rd value of the 25th data instance: 1.900
```

Iris data set we have used above has four continous attributes. Here's a script that computes their mean:

```
average = lambda x: sum(x)/len(x)

data = Orange.data.Table("iris")
print("%-15s %s" % ("Feature", "Mean"))
for x in data.domain.attributes:
    print("%-15s %.2f" % (x.name, average([d[x] for d in data])))
```

Above also illustrates indexing of data instances with objects that store features; in `d[x]` variable `x` is an Orange object. Here's the output:

```
Feature         Mean
sepal length    5.84
sepal width     3.05
petal length    3.76
petal width     1.20
```

A slightly more complicated, but more interesting is a code that computes per-class averages:

```
average = lambda xs: sum(xs)/float(len(xs))

data = Orange.data.Table("iris")
```

```
targets = data.domain.class_var.values
print("%-15s %s" % ("Attribute", " ".join("%15s" % c for c in targets)))
for a in data.domain.attributes:
    dist = ["%15.2f" % average([d[a] for d in data if d.get_class() == c])
            for c in targets]
    print("%-15s" % a.name, " ".join(dist))
```

Of the four features, petal width and length look quite discriminative for the type of iris:

Feature Iris-setosa Iris-versicolor Iris-virginica sepal length 5.01 5.94 6.59 sepal width 3.42 2.77 2.97 petal length 1.46 4.26 5.55 petal width 0.24 1.33 2.03

Finally, here is a quick code that computes the class distribution for another data set:

```
import Orange
from collections import Counter

data = Orange.data.Table("lenses")
print(Counter(str(d.get_class()) for d in data))
```

Orange Data Sets and NumPy

Orange data sets are actually wrapped NumPy arrays. Wrapping is performed to retain the information about the feature names and values, and NumPy arrays are used for speed and compatibility with different machine learning toolboxes, like scikit-learn, on which Orange relies. Let us display the values of these arrays for the first three data instances of the iris data set:

```
>>> data = Orange.data.Table("iris")
>>> data.X[:3]
array([[ 5.1,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2]])
>>> data.Y[:3]
array([ 0.,  0.,  0.])
```

Notice that we access the arrays for attributes and class separately, using `data.X` and `data.Y`. Average values of attributes can then be computed efficiently by:

```
>>> import np as numpy
>>> np.mean(data.X, axis=0)
array([ 5.84333333,  3.054     ,  3.75866667,  1.19866667])
```

We can also construct a (classless) data set from a numpy array:

```
>>> X = np.array([[1,2], [4,5]])
>>> data = Orange.data.Table(X)
>>> data.domain
[Feature 1, Feature 2]
```

If we want to provide meaninful names to attributes, we need to construct an appropriate data domain:

```
>>> domain = Orange.data.Domain([Orange.data.ContinuousVariable("lenght"),
                                 Orange.data.ContinuousVariable("width")])
>>> data = Orange.data.Table(domain, X)
>>> data.domain
[lenght, width]
```

Here is another example, this time with construction of data set that includes a numerical class and different type of attributes:

```
size = Orange.data.DiscreteVariable("size", ["small", "big"])
height = Orange.data.ContinuousVariable("height")
shape = Orange.data.DiscreteVariable("shape", ["circle", "square", "oval"])
speed = Orange.data.ContinuousVariable("speed")

domain = Orange.data.Domain([size, height, shape], speed)

X = np.array([[1, 3.4, 0], [0, 2.7, 2], [1, 1.4, 1]])
Y = np.array([42.0, 52.2, 13.4])

data = Orange.data.Table(domain, X, Y)
print(data)
```

Running of this scripts yields:

```
[[big, 3.400, circle | 42.000],
 [small, 2.700, oval | 52.200],
 [big, 1.400, square | 13.400]
```

Meta Attributes

Often, we wish to include descriptive fields in the data that will not be used in any computation (distance estimation, modeling), but will serve for identification or additional information. These are called meta attributes, and are marked with `meta` in the third header row:

```
name        hair        eggs        milk        backbone        legs        type
string        d        d        d        d        d        d
meta                                                    class
aardvark        1        0        1        1        4        mammal
antelope        1        0        1        1        4        mammal
bass        0        1        0        1        0        fish
bear        1        0        1        1        4        mammal
```

Values of meta attributes and all other (non-meta) attributes are treated similarly in Orange, but stored in the separate numpy arrays:

```
>>> data = Orange.data.Table("zoo")
>>> data[0]["name"]
>>> data[0]["type"]
>>> for d in data:
    ...:     print("{}/{}: {}".format(d["name"], d["type"], d["legs"]))
    ...:
aardvark/mammal: 4
antelope/mammal: 4
bass/fish: 0
bear/mammal: 4
>>> data.X
array([[ 1.,  0.,  1.,  1.,  2.],
       [ 1.,  0.,  1.,  1.,  2.],
       [ 0.,  1.,  0.,  1.,  0.],
       [ 1.,  0.,  1.,  1.,  2.]]))
>>> data.metas
array([['aardvark'],
       ['antelope'],
```

```
       ['bass'],
       ['bear']], dtype=object))
```

Meta attributes may be passed to `Orange.data.Table` after providing arrays for attribute and class values:

```
from Orange.data import Table, Domain
from Orange.data import ContinuousVariable, DiscreteVariable, StringVariable
import numpy as np

X = np.array([[2.2, 1625], [0.3, 163]])
Y = np.array([0, 1])
M = np.array([["houston", 10], ["ljubljana", -1]])

domain = Domain([ContinuousVariable("population"), ContinuousVariable("area")],
                [DiscreteVariable("snow", ("no", "yes"))],
                [StringVariable("city"), StringVariable("temperature")])
data = Table(domain, X, Y, M)
print(data)
```

The script outputs:

```
[[2.200, 1625.000 | no] {houston, 10},
 [0.300, 163.000 | yes] {ljubljana, -1}
```

To construct a classless domain we could pass `None` for the class values.

Missing Values

Consider the following exploration of the data set on votes of the US senate:

```
>>> import numpy as np
>>> data = Orange.data.Table("voting.tab")
>>> data[2]
[?, y, y, ?, y, ... | democrat]
>>> np.isnan(data[2][0])
True
>>> np.isnan(data[2][1])
False
```

The particular data instance included missing data (represented with '?') for the first and the fourth attribute. In the original data set file, the missing values are, by default, represented with a blank space. We can now examine each attribute and report on proportion of data instances for which this feature was undefined:

```
data = Orange.data.Table("voting.tab")
for x in data.domain.attributes:
    n_miss = sum(1 for d in data if np.isnan(d[x]))
    print("%4.1f%% %s" % (100.*n_miss/len(data), x.name))
```

First three lines of the output of this script are:

```
 2.8% handicapped-infants
11.0% water-project-cost-sharing
 2.5% adoption-of-the-budget-resolution
```

A single-liner that reports on number of data instances with at least one missing value is:

```
>>> sum(any(np.isnan(d[x]) for x in data.domain.attributes) for d in data)
203
```

Data Selection and Sampling

Besides the name of the data file, `Orange.data.Table` can accept the data domain and a list of data items and returns a new data set. This is useful for any data subsetting:

```
data = Orange.data.Table("iris.tab")
print("Data set instances:", len(data))
subset = Orange.data.Table(data.domain,
                           [d for d in data if d["petal length"] > 3.0])
print("Subset size:", len(subset))
```

The code outputs:

```
Data set instances: 150
Subset size: 99
```

and inherits the data description (domain) from the original data set. Changing the domain requires setting up a new domain descriptor. This feature is useful for any kind of feature selection:

```
data = Orange.data.Table("iris.tab")
new_domain = Orange.data.Domain(list(data.domain.attributes[:2]),
                                data.domain.class_var)
new_data = Orange.data.Table(new_domain, data)

print(data[0])
print(new_data[0])
```

We could also construct a random sample of the data set:

```
>>> sample = Orange.data.Table(data.domain, random.sample(data, 3))
>>> sample
[[6.000, 2.200, 4.000, 1.000 | Iris-versicolor],
 [4.800, 3.100, 1.600, 0.200 | Iris-setosa],
 [6.300, 3.400, 5.600, 2.400 | Iris-virginica]
]
```

or randomly sample the attributes:

```
>>> atts = random.sample(data.domain.attributes, 2)
>>> domain = Orange.data.Domain(atts, data.domain.class_var)
>>> new_data = Orange.data.Table(domain, data)
>>> new_data[0]
[5.100, 1.400 | Iris-setosa]
```

Classification

Much of Orange is devoted to machine learning methods for classification, or supervised data mining. These methods rely on the data with class-labeled instances, like that of senate voting. Here is a code that loads this data set, displays the first data instance and shows its predicted class (`republican`):

```
>>> import Orange
>>> data = Orange.data.Table("voting")
>>> data[0]
[n, y, n, y, y, ... | republican]
```

Orange implements function for construction of classification models, their evaluation and scoring. In a nutshel, here is the code that reports on cross-validated accuracy and AUC for logistic regression and random forests:

```
import Orange

data = Orange.data.Table("voting")
lr = Orange.classification.LogisticRegressionLearner()
rf = Orange.classification.RandomForestLearner(n_estimators=100)
res = Orange.evaluation.CrossValidation(data, [lr, rf], k=5)

print("Accuracy:", Orange.evaluation.scoring.CA(res))
print("AUC:", Orange.evaluation.scoring.AUC(res))
```

It turns out that for this domain logistic regression does well:

```
Accuracy: [ 0.96321839  0.95632184]
AUC: [ 0.96233796  0.95671252]
```

For supervised learning, Orange uses learners. These are objects that recieve the data and return classifiers. Learners are passed to evaluation rutines, such as cross-validation above.

Learners and Classifiers

Classification uses two types of objects: learners and classifiers. Learners consider class-labeled data and return a classifier. Given the first three data instances, classifiers return the indexes of predicted class:

```
>>> import Orange
>>> data = Orange.data.Table("voting")
>>> learner = Orange.classification.LogisticRegressionLearner()
>>> classifier = learner(data)
>>> classifier(data[:3])
array([ 0.,  0.,  1.])
```

Above, we read the data, constructed a logistic regression learner, gave it the data set to construct a classifier, and used it to predict the class of the first three data instances. We also use these concepts in the following code that predicts the classes of the selected three instances in the data set:

```
learner = Orange.classification.LogisticRegressionLearner()
classifier = learner(data)
c_values = data.domain.class_var.values
for d in data[5:8]:
    c = classifier(d)
    print("{}, originally {}".format(c_values[int(classifier(d)[0])],
                                     d.get_class()))
```

The script outputs:

```
democrat, originally democrat
republican, originally democrat
republican, originally republican
```

Logistic regression has made a mistake in the second case, but otherwise predicted correctly. No wonder, since this was also the data it trained from. The following code counts the number of such mistakes in the entire data set:

```
data = Orange.data.Table("voting")
learner = Orange.classification.LogisticRegressionLearner()
classifier = learner(data)
x = np.sum(data.Y != classifier(data))
```

Probabilistic Classification

To find out what is the probability that the classifier assigns to, say, democrat class, we need to call the classifier with additional parameter that specifies the classification output type.

```
data = Orange.data.Table("voting")
learner = Orange.classification.LogisticRegressionLearner()
classifier = learner(data)
target_class = 1
print("Probabilities for %s:" % data.domain.class_var.values[target_class])
probabilities = classifier(data, 1)
for p, d in zip(probabilities[5:8], data[5:8]):
    print(p[target_class], d.get_class())
```

The output of the script also shows how badly the logistic regression missed the class in the second case:

```
Probabilities for democrat:
0.999506847581 democrat
0.201139534658 democrat
0.042347504805 republican
```

Cross-Validation

Validating the accuracy of classifiers on the training data, as we did above, serves demonstration purposes only. Any performance measure that assess accuracy should be estimated on the independent test set. Such is also a procedure called cross-validation, which averages the evaluation scores across several runs, each time considering a different training and test subsets as sampled from the original data set:

```
data = Orange.data.Table("titanic")
lr = Orange.classification.LogisticRegressionLearner()
res = Orange.evaluation.CrossValidation(data, [lr], k=5)
print("Accuracy: %.3f" % Orange.evaluation.scoring.CA(res)[0])
print("AUC:      %.3f" % Orange.evaluation.scoring.AUC(res)[0])
```

Cross-validation is expecting a list of learners. The performance estimators also return a list of scores, one for every learner. There was just one learner (*lr*) in the script above, hence the array of length one was return. The script estimates classification accuracy and area under ROC curve:

```
Accuracy: 0.779
AUC:      0.704
```

Handful of Classifiers

Orange includes a variety of classification algorithms, most of them wrapped from scikit-learn, including:

- logistic regression (`Orange.classification.LogisticRegressionLearner`)
- k-nearest neighbors (`Orange.classification.knn.KNNLearner`)
- support vector machines (say, `Orange.classification.svm.LinearSVMLearner`)
- classification trees (`Orange.classification.tree.SklTreeLearner`)
- radnom forest (`Orange.classification.RandomForestLearner`)

Some of these are included in the code that estimates the probability of a target class on a testing data. This time, training and test data sets are disjoint:

```
import Orange
import random

random.seed(42)
data = Orange.data.Table("voting")
test = Orange.data.Table(data.domain, random.sample(data, 5))
train = Orange.data.Table(data.domain, [d for d in data if d not in test])

tree = Orange.classification.tree.TreeLearner(max_depth=3)
knn = Orange.classification.knn.KNNLearner(n_neighbors=3)
lr = Orange.classification.LogisticRegressionLearner(C=0.1)

learners = [tree, knn, lr]
classifiers = [learner(train) for learner in learners]

target = 0
print("Probabilities for %s:" % data.domain.class_var.values[target])
print("original class ", " ".join("%-5s" % l.name for l in classifiers))

c_values = data.domain.class_var.values
for d in test:
    print(("{:<15}" + " {:.3f}"*len(classifiers)).format(
        c_values[int(d.get_class())],
        *(c(d, 1)[0][target] for c in classifiers)))
```

For these five data items, there are no major differences between predictions of observed classification algorithms:

```
Probabilities for republican:
original class  tree  knn   logreg
republican      0.991 1.000 0.966
republican      0.991 1.000 0.985
democrat        0.000 0.000 0.021
republican      0.991 1.000 0.979
republican      0.991 0.667 0.963
```

The following code cross-validates these learners on the titanic data set.

```
import Orange

data = Orange.data.Table("titanic")
tree = Orange.classification.tree.TreeLearner(max_depth=3)
knn = Orange.classification.knn.KNNLearner(n_neighbors=3)
lr = Orange.classification.LogisticRegressionLearner(C=0.1)
learners = [tree, knn, lr]

print(" "*9 + " ".join("%-4s" % learner.name for learner in learners))
res = Orange.evaluation.CrossValidation(data, learners, k=5)
```

```
print("Accuracy %s" % " ".join("%.2f" % s for s in Orange.evaluation.CA(res)))
print("AUC      %s" % " ".join("%.2f" % s for s in Orange.evaluation.AUC(res)))
```

Logistic regression wins in area under ROC curve:

```
         tree knn  logreg
Accuracy 0.79 0.47 0.78
AUC      0.68 0.56 0.70
```

Regression

Regression in Orange is from the interface very similar to classification. These both require class-labeled data. Just like in classification, regression is implemented with learners and regression models (regressors). Regression learners are objects that accept data and return regressors. Regression models are given data items to predict the value of continuous class:

```
import Orange

data = Orange.data.Table("housing")
learner = Orange.regression.LinearRegressionLearner()
model = learner(data)

print("predicted, observed:")
for d in data[:3]:
    print("%.1f, %.1f" % (model(d)[0], d.get_class()))
```

Handful of Regressors

Let us start with regression trees. Below is an example script that builds the tree from data on housing prices and prints out the tree in textual form:

```
data = Orange.data.Table("housing")
tree_learner = Orange.regression.SimpleTreeLearner(max_depth=2)
tree = tree_learner(data)
print(tree.to_string())
```

The script outputs the tree:

```
RM<=6.941: 19.9
RM>6.941
|    RM<=7.437
|    |    CRIM>7.393: 14.4
|    |    CRIM<=7.393
|    |    |    DIS<=1.886: 45.7
|    |    |    DIS>1.886: 32.7
|    RM>7.437
|    |    TAX<=534.500: 45.9
|    |    TAX>534.500: 21.9
```

Following is initialization of few other regressors and their prediction of the first five data instances in housing price data set:

```
random.seed(42)
data = Orange.data.Table("housing")
test = Orange.data.Table(data.domain, random.sample(data, 5))
train = Orange.data.Table(data.domain, [d for d in data if d not in test])

lin = Orange.regression.linear.LinearRegressionLearner()
rf = Orange.regression.random_forest.RandomForestRegressionLearner()
rf.name = "rf"
ridge = Orange.regression.RidgeRegressionLearner()

learners = [lin, rf, ridge]
regressors = [learner(train) for learner in learners]

print("y    ", " ".join("%5s" % l.name for l in regressors))

for d in test:
    print(("{:<5}" + " {:5.1f}"*len(regressors)).format(
        d.get_class(),
        *(r(d)[0] for r in regressors)))
```

Looks like the housing prices are not that hard to predict:

```
y    linreg    rf ridge
22.2   19.3  21.8  19.5
31.6   33.2  26.5  33.2
21.7   20.9  17.0  21.0
10.2   16.9  14.3  16.8
14.0   13.6  14.9  13.5
```

Cross Validation

Evaluation and scoring methods are available at `Orange.evaluation`:

```
data = Orange.data.Table("housing.tab")

lin = Orange.regression.linear.LinearRegressionLearner()
rf = Orange.regression.random_forest.RandomForestRegressionLearner()
rf.name = "rf"
ridge = Orange.regression.RidgeRegressionLearner()
mean = Orange.regression.MeanLearner()

learners = [lin, rf, ridge, mean]

res = Orange.evaluation.CrossValidation(data, learners, k=5)
rmse = Orange.evaluation.RMSE(res)
r2 = Orange.evaluation.R2(res)

print("Learner  RMSE  R2")
for i in range(len(learners)):
    print("{:8s} {:.2f} {:5.2f}".format(learners[i].name, rmse[i], r2[i]))
```

We have scored the regression two measures for goodnes of fit: root-mean-square error and coefficient of determination, or R squared. Random forest has the lowest root mean squared error:

```
Learner  RMSE  R2
linreg   4.88  0.72
rf       4.70  0.74
ridge    4.91  0.71
mean     9.20 -0.00
```

Not much difference here. Each regression method has a set of parameters. We have been running them with default parameters, and parameter fitting would help. Also, we have included `MeanLearner` in a list of our regression; this regressors simply predicts the mean value from the training set, and is used as a baseline.

CHAPTER 2

Reference

Available classes and methods.

Data model (`data`)

Orange stores data in `Orange.data.Storage` classes. The most commonly used storage is *Orange.data.Table*, which stores all data in two-dimensional numpy arrays. Each row of the data represents a data instance.

Individual data instances are represented as instances of *Orange.data.Instance*. Different storage classes may derive subclasses of *Instance* to represent the retrieved rows in the data more efficiently and to allow modifying the data through modifying data instance. For example, if *table* is *Orange.data.Table*, *table[0]* returns the row as *Orange.data.RowInstance*.

Every storage class and data instance has an associated domain description *domain* (an instance of *Orange.data.Domain*) that stores descriptions of data columns. Every column is described by an instance of a class derived from *Orange.data.Variable*. The subclasses correspond to continuous variables (*Orange.data.ContinuousVariable*), discrete variables (*Orange.data.DiscreteVariable*) and string variables (*Orange.data.StringVariable*). These descriptors contain the variable's name, symbolic values, number of decimals in printouts and similar.

The data is divided into attributes (features, independent variables), class variables (classes, targets, outcomes, dependent variables) and meta attributes. This division applies to domain descriptions, data storages that contain separate arrays for each of the three parts of the data and data instances.

Attributes and classes are represented with numeric values and are used in modelling. Meta attributes contain additional data which may be of any type. (Currently, only string values are supported in addition to continuous and numeric.)

In indexing, columns can be referred to by their names, descriptors or an integer index. For example, if *inst* is a data instance and *var* is a descriptor of type `Continuous`, referring to the first column in the data, which is also names "petal length", then *inst[var]*, *inst[0]* and *inst["petal length"]* refer to the first value of the instance. Negative indices are used for meta attributes, starting with -1.

Continuous and discrete values can be represented by any numerical type; by default, Orange uses double precision (64-bit) floats. Discrete values are represented by whole numbers.

Data Storage (`storage`)

`Orange.data.storage.Storage` is an abstract class representing a data object in which rows represent data instances (examples, in machine learning terminology) and columns represent variables (features, attributes, classes, targets, meta attributes).

Data is divided into three parts that represent independent variables (*X*), dependent variables (*Y*) and meta data (*metas*). If practical, the class should expose those parts as properties. In the associated domain (*Orange.data.Domain*), the three parts correspond to lists of variable descriptors *attributes*, *class_vars* and *metas*.

Any of those parts may be missing, dense, sparse or sparse boolean. The difference between the later two is that the sparse data can be seen as a list of pairs (variable, value), while in the latter the variable (item) is present or absent, like in market basket analysis. The actual storage of sparse data depends upon the storage type.

There is no uniform constructor signature: every derived class provides one or more specific constructors.

There are currently two derived classes *Orange.data.Table* and `Orange.data.sql.Table`, the former storing the data in-memory, in numpy objects, and the latter in SQL (currently, only PostreSQL is supported).

Derived classes must implement at least the methods for getting rows and the number of instances (*__getitem__* and *__len__*). To make storage fast enough to be practically useful, it must also reimplement a number of filters, preprocessors and aggregators. For instance, method *_filter_values(self, filter)* returns a new storage which only contains the rows that match the criteria given in the filter. *Orange.data.Table* implements an efficient method based on numpy indexing, and `Orange.data.sql.Table`, which "stores" a table as an SQL query, converts the filter into a WHERE clause.

`Orange.data.storage.`**`domain`** (*:obj:'Orange.data.Domain'*)
The domain describing the columns of the data

Data access

`Orange.data.storage.`**`__getitem__`**(*self*, *index*)
Return one or more rows of the data.

- If the index is an int, e.g. *data[7]*; the corresponding row is returned as an instance of `Instance`. Concrete implementations of *Storage* use specific derived classes for instances.
- If the index is a slice or a sequence of ints (e.g. *data[7:10]* or *data[[7, 42, 15]]*, indexing returns a new storage with the selected rows.
- If there are two indices, where the first is an int (a row number) and the second can be interpreted as columns, e.g. *data[3, 5]* or *data[3, 'gender']* or *data[3, y]* (where *y* is an instance of *Variable*), a single value is returned as an instance of `Value`.
- In all other cases, the first index should be a row index, a slice or a sequence, and the second index, which represent a set of columns, should be an int, a slice, a sequence or a numpy array. The result is a new storage with a new domain.

`.__len__`(*self*)
Return the number of data instances (rows)

Inspection

`Storage.X_density, Storage.Y_density, Storage.metas_density`
Indicates whether the attributes, classes and meta attributes are dense (*Storage.DENSE*) or sparse (*Storage.SPARSE*). If they are sparse and all values are 0 or 1, it is marked as (*Storage.SPARSE_BOOL*). The Storage class provides a default DENSE. If the data has no attibutes, classes or meta attributes, the corresponding method should re

Filters

Storage should define the following methods to optimize the filtering operations as allowed by the underlying data structure. *Orange.data.Table* executes them directly through numpy (or bottleneck or related) methods, while Orange.data.sql.Table appends them to the WHERE clause of the query that defines the data.

These methods should not be called directly but through the classes defined in *Orange.data.filter*. Methods in *Orange.data.filter* also provide the slower fallback functions for the functions not defined in the storage.

Orange.data.storage.**_filter_is_defined**(*self*, *columns=None*, *negate=False*)
Extract rows without undefined values.

Parameters

- **columns** (*sequence of ints, variable names or descriptors*) – optional list of columns that are checked for unknowns
- **negate** (*bool*) – invert the selection

Returns a new storage of the same type or *Table*

Return type Orange.data.storage.Storage

Orange.data.storage.**_filter_has_class**(*self*, *negate=False*)
Return rows with known value of the target attribute. If there are multiple classes, all must be defined.

Parameters **negate** (*bool*) – invert the selection

Returns a new storage of the same type or *Table*

Return type Orange.data.storage.Storage

Orange.data.storage.**_filter_same_value**(*self*, *column*, *value*, *negate=False*)
Select rows based on a value of the given variable.

Parameters

- **column** (*int, str or* Orange.data.Variable) – the column that is checked
- **value** (*int, float or str*) – the value of the variable
- **negate** (*bool*) – invert the selection

Returns a new storage of the same type or *Table*

Return type Orange.data.storage.Storage

Orange.data.storage.**_filter_values**(*self*, *filter*)
Apply a the given filter to the data.

Parameters **filter** (*Orange.data.Filter*) – A filter for selecting the rows

Returns a new storage of the same type or *Table*

Return type Orange.data.storage.Storage

Aggregators

Similarly to filters, storage classes should provide several methods for fast computation of statistics. These methods are not called directly but by modules within Orange.statistics.

_compute_basic_stats(
self, columns=None, include_metas=False, compute_variance=False)
Compute basic statistics for the specified variables: minimal and maximal value, the mean and a varianca (or a zero placeholder), the number of missing and defined values.

Parameters

- **columns** (list of ints, variable names or descriptors of type *Orange.data.Variable*) – a list of columns for which the statistics is computed; if *None*, the function computes the data for all variables
- **include_metas** (*bool*) – a flag which tells whether to include meta attributes (applicable only if *columns* is *None*)
- **compute_variance** (*bool*) – a flag which tells whether to compute the variance

Returns a list with tuple (min, max, mean, variance, #nans, #non-nans) for each variable

Return type *list*

Orange.data.storage.**_compute_distributions**(*self*, *columns=None*)

Compute the distribution for the specified variables. The result is a list of pairs containing the distribution and the number of rows for which the variable value was missing.

For discrete variables, the distribution is represented as a vector with absolute frequency of each value. For continuous variables, the result is a 2-d array of shape (2, number-of-distinct-values); the first row contains (distinct) values of the variables and the second has their absolute frequencies.

Parameters **columns** (list of ints, variable names or descriptors of type *Orange.data.Variable*) – a list of columns for which the distributions are computed; if *None*, the function runs over all variables

Returns a list of distributions

Return type list of numpy arrays

Storage.**_compute_contingency**(*col_vars=None*, *row_var=None*)

Compute contingency matrices for one or more discrete or continuous variables against the specified discrete variable.

The resulting list contains a pair for each column variable. The first element contains the contingencies and the second elements gives the distribution of the row variables for instances in which the value of the column variable is missing.

The format of contingencies returned depends on the variable type:

•for discrete variables, it is a numpy array, where element (i, j) contains count of rows with i-th value of the row variable and j-th value of the column variable.

•for continuous variables, contingency is a list of two arrays, where the first array contains ordered distinct values of the column_variable and the element (i,j) of the second array contains count of rows with i-th value of the row variable and j-th value of the ordered column variable.

Parameters

- **col_vars** (list of ints, variable names or descriptors of type *Orange.data.Variable*) – variables whose values will correspond to columns of contingency matrices
- **row_var** (int, variable name or *Orange.data.DiscreteVariable*) – a discrete variable whose values will correspond to the rows of contingency matrices

Data Table (table)

class Orange.data.**Table**(**args*, ***kwargs*)

Stores data instances as a set of 2d tables representing the independent variables (attributes, features) and dependent variables (classes, targets), and the corresponding weights and meta attributes.

The data is stored in 2d numpy arrays `X`, `Y`, `W`, `metas`. The arrays may be dense or sparse. All arrays have the same number of rows. If certain data is missing, the corresponding array has zero columns.

Arrays can be of any type; default is *float* (that is, double precision). Values of discrete variables are stored as whole numbers. Arrays for meta attributes usually contain instances of *object*.

The table also stores the associated information about the variables as an instance of *Domain*. The number of columns must match the corresponding number of variables in the description.

There are multiple ways to get values or entire rows of the table.

- The index can be an int, e.g. *table[7]*; the corresponding row is returned as an instance of *RowInstance*.
- The index can be a slice or a sequence of ints (e.g. *table[7:10]* or *table[[7, 42, 15]]*, indexing returns a new data table with the selected rows.
- If there are two indices, where the first is an int (a row number) and the second can be interpreted as columns, e.g. *table[3, 5]* or *table[3, 'gender']* or *table[3, y]* (where *y* is an instance of *Variable*), a single value is returned as an instance of `Value`.
- In all other cases, the first index should be a row index, a slice or a sequence, and the second index, which represent a set of columns, should be an int, a slice, a sequence or a numpy array. The result is a new table with a new domain.

Rules for setting the data are as follows.

- If there is a single index (an *int*, *slice*, or a sequence of row indices) and the value being set is a single scalar, all attributes (not including the classes) are set to that value. That is, *table[r] = v* is equivalent to *table.X[r] = v*.
- If there is a single index and the value is a data instance (*Orange.data.Instance*), it is converted into the table's domain and set to the corresponding rows.
- Final option for a single index is that the value is a sequence whose length equals the number of attributes and target variables. The corresponding rows are set; meta attributes are set to unknowns.
- **For two indices, the row can again be given as a single *int*, a** *slice* or a sequence of indices. Column indices can be a single *int*, *str* or *Orange.data.Variable*, a sequence of them, a *slice* or any iterable. The value can be a single value, or a sequence of appropriate length.

domain
Description of the variables corresponding to the table's columns. The domain is used for determining the variable types, printing the data in human-readable form, conversions between data tables and similar.

columns
A class whose attributes contain attribute descriptors for columns. For a table *table*, setting *c = table.columns* will allow accessing the table's variables with, for instance *c.gender*, *c.age* ets. Spaces are replaced with underscores.

Constructors

The preferred way to construct a table is to invoke a named constructor.

classmethod `Table.`**from_domain**(*domain*, *n_rows=0*, *weights=False*)
Construct a new *Table* with the given number of rows for the given domain. The optional vector of weights is initialized to 1's.

Parameters

- **domain** (*Orange.data.Domain*) – domain for the *Table*
- **n_rows** (*int*) – number of rows in the new table

- **weights** (*bool*) – indicates whether to construct a vector of weights

Returns a new table

Return type *Orange.data.Table*

classmethod Table.**from_table**(*domain*, *source*, *row_indices=Ellipsis*)

Create a new table from selected columns and/or rows of an existing one. The columns are chosen using a domain. The domain may also include variables that do not appear in the source table; they are computed from source variables if possible.

The resulting data may be a view or a copy of the existing data.

Parameters

- **domain** (Orange.data.Domain) – the domain for the new table
- **source** (Orange.data.Table) – the source table
- **row_indices** (*a slice or a sequence*) – indices of the rows to include

Returns a new table

Return type *Orange.data.Table*

classmethod Table.**from_table_rows**(*source*, *row_indices*)

Construct a new table by selecting rows from the source table.

Parameters

- **source** (Orange.data.Table) – an existing table
- **row_indices** (*a slice or a sequence*) – indices of the rows to include

Returns a new table

Return type *Orange.data.Table*

classmethod Table.**from_numpy**(*domain*, *X*, *Y=None*, *metas=None*, *W=None*)

Construct a table from numpy arrays with the given domain. The number of variables in the domain must match the number of columns in the corresponding arrays. All arrays must have the same number of rows. Arrays may be of different numpy types, and may be dense or sparse.

Parameters

- **domain** (Orange.data.Domain) – the domain for the new table
- **X** (*np.array*) – array with attribute values
- **Y** (*np.array*) – array with class values
- **metas** (*np.array*) – array with meta attributes
- **W** (*np.array*) – array with weights

Returns

classmethod Table.**from_file**(*filename*, *sheet=None*)

Read a data table from a file. The path can be absolute or relative.

Parameters

- **filename** (*str*) – File name
- **sheet** (*str*) – Sheet in a file (optional)

Returns a new data table

Return type *Orange.data.Table*

Inspection

Table.**is_view**()
Return *True* if all arrays represent a view referring to another table

Table.**is_copy**()
Return *True* if the table owns its data

Table.**ensure_copy**()
Ensure that the table owns its data; copy arrays when necessary.

Table.**has_missing**()
Return *True* if there are any missing attribute or class values.

Table.**has_missing_class**()
Return *True* if there are any missing class values.

Table.**checksum**(*include_metas=True*)
Return a checksum over X, Y, metas and W.

Row manipulation

Table.**append**(*instance*)
Append a data instance to the table.

Parameters **instance** (*Orange.data.Instance or a sequence of values*) – a data instance

Table.**extend**(*instances*)
Extend the table with the given instances. The instances can be given as a table of the same or a different domain, or a sequence. In the latter case, each instances can be given as *Instance* or a sequence of values (e.g. list, tuple, numpy.array).

Parameters **instances** (*Orange.data.Table or a sequence of instances*) – additional instances

Table.**insert**(*row*, *instance*)
Insert a data instance into the table.

Parameters

- **row** (*int*) – row index
- **instance** (*Orange.data.Instance or a sequence of values*) – a data instance

Table.**clear**()
Remove all rows from the table.

Table.**shuffle**()
Randomly shuffle the rows of the table.

Weights

Table.**has_weights**()
Return *True* if the data instances are weighed.

Table.**set_weights**(*weight=1*)
Set weights of data instances; create a vector of weights if necessary.

Table.**total_weight**()
Return the total weight of instances in the table, or their number if they are unweighted.

SQL table (data.sql)

class Orange.data.sql.table.**SqlTable**(*connection_params*, *table_or_sql*, *backend=None*, *type_hints=None*, *inspect_values=False*)
SqlTable represents a table with the data which is stored in the database. Besides the inherited attributes, the object stores a connection to the database and row filters.

Constructor connects to the database, infers the variable types from the types of the columns in the database and constructs the corresponding domain description. Discrete and continuous variables are put among attributes, and string variables are meta attributes. The domain does not have a class.

SqlTable overloads the data access methods for random access to rows and for iteration (*__getitem__* and *__iter__*). It also provides methods for fast computation of basic statistics, distributions and contingency matrices, as well as for filtering the data. Filtering the data returns a new instance of *SqlTable*. The new instances however differs only in that an additional filter is added to the row_filter.

All evaluation is lazy in the sense that most operations just modify the domain and the list of filters. These are used to construct an SQL query when the data is actually needed, for instance to retrieve a data row or compute a distribution of values for a certain column.

connection
The object that holds the database connection. An instance of a class compatible with Python DB API 2.0.

host
The host name of the database server

database
The name of the database

table_name
The name of the table in the database

row_filters
A list of filters that are applied when constructing the query. The filters in the should have a method *to_sql*. Module Orange.data.sql.filter contains classes derived from filters in *Orange.data.filter* with the appropriate implementation of the method.

__init__(*connection_params*, *table_or_sql*, *backend=None*, *type_hints=None*, *inspect_values=False*)
Create a new proxy for sql table.

To create a new SqlTable, specify the connection parameters for psycopg2 and the name of the table/sql query used to fetch the data.

table = SqlTable('database_name', 'table_name') table = SqlTable('database_name', 'SELECT * FROM table')

For complex configurations, dictionary of connection parameters can be used instead of the database name. For documentation about connection parameters, see: http://www.postgresql.org/docs/current/static/libpq-connect.html#LIBPQ-PARAMKEYWORDS

Data domain is inferred from the columns of the table/query.

The (very quick) default setting is to treat all numeric columns as continuous variables and everything else as strings and placed among meta attributes.

If inspect_values parameter is set to True, all column values are inspected and int/string columns with less than 21 values are intepreted as discrete features.

Domains can be constructed by the caller and passed in type_hints parameter. Variables from the domain are used for the columns with the matching names; for columns without the matching name in the domain, types are inferred as described above.

__getitem__(*key*)
Indexing of SqlTable is performed in the following way:

If a single row is requested, it is fetched from the database and returned as a SqlRowInstance.

A new SqlTable with appropriate filters is constructed and returned otherwise.

__iter__()
Iterating through the rows executes the query using a cursor and then yields resulting rows as SqlRowInstances as they are requested.

copy()
Return a copy of the SqlTable

__bool__()
Return True if the SqlTable is not empty.

__len__()
Return number of rows in the table. The value is cached so it is computed only the first time the length is requested.

download_data(*limit=None*, *partial=False*)
Download SQL data and store it in memory as numpy matrices.

X
Numpy array with attribute values.

Y
Numpy array with class values.

metas
Numpy array with class values.

W
Numpy array with class values.

ids
Numpy array with class values.

class Orange.data.sql.table.**SqlRowInstance**(*domain*, *data=None*)
Extends *Orange.data.Instance* to correctly handle values of meta attributes.

Domain description (domain)

Description of a domain stores a list of features, class(es) and meta attribute descriptors. A domain descriptor is attached to all tables in Orange to assign names and types to the corresponding columns. Columns in the *Orange.data.Table* have the roles of attributes (features, independent variables), class(es) (targets, outcomes, dependent variables) and meta attributes; in parallel to that, the domain descriptor stores their corresponding descriptions in collections of variable descriptors of type *Orange.data.Variable*.

Domain descriptors are also stored in predictive models and other objects to facilitate automated conversions between domains, as described below.

Domains are most often constructed automatically when loading the data or wrapping the numpy arrays into Orange's *Table*.

```
>>> from Orange.data import Table
>>> iris = Table("iris")
>>> iris.domain
[sepal length, sepal width, petal length, petal width | iris]
```

class Orange.data.**Domain**(*attributes*, *class_vars=None*, *metas=None*, *source=None*)

attributes
A tuple of descriptors (instances of *Orange.data.Variable*) for attributes (features, independent variables).

```
>>> iris.domain.attributes
(ContinuousVariable('sepal length'), ContinuousVariable('sepal width'),
ContinuousVariable('petal length'), ContinuousVariable('petal width'))
```

class_var
Class variable if the domain has a single class; *None* otherwise.

```
>>> iris.domain.class_var
DiscreteVariable('iris')
```

class_vars
A tuple of descriptors for class attributes (outcomes, dependent variables).

```
>>> iris.domain.class_vars
(DiscreteVariable('iris'),)
```

variables
A list of attributes and class attributes (the concatenation of the above).

```
>>> iris.domain.variables
(ContinuousVariable('sepal length'), ContinuousVariable('sepal width'),
ContinuousVariable('petal length'), ContinuousVariable('petal width'),
DiscreteVariable('iris'))
```

metas
List of meta attributes.

anonymous
True if the domain was constructed when converting numpy array to *Orange.data.Table*. Such domains can be converted to and from other domains even if they consist of different variable descriptors for as long as their number and types match.

__init__(*attributes*, *class_vars=None*, *metas=None*, *source=None*)
Initialize a new domain descriptor. Arguments give the features and the class attribute(s). They can be described by descriptors (instances of *Variable*), or by indices or names if the source domain is given.

Parameters

- **attributes** (list of *Variable*) – a list of attributes
- **class_vars** (*Variable* or list of *Variable*) – target variable or a list of target variables
- **metas** (list of *Variable*) – a list of meta attributes
- **source** (Orange.data.Domain) – the source domain for attributes

Returns a new domain

Return type *Domain*

The following script constructs a domain with a discrete feature *gender* and continuous feature *age*, and a continuous target *salary*.

```
>>> from Orange.data import Domain, DiscreteVariable, ContinuousVariable
>>> domain = Domain([DiscreteVariable.make("gender"),
...                  ContinuousVariable.make("age")],
...                 ContinuousVariable.make("salary"))
>>> domain
[gender, age | salary]
```

This constructs a new domain with some features from the Iris data set and a new feature *color*.

```
>>> new_domain = Domain(["sepal length",
...                      "petal length",
...                      DiscreteVariable.make("color")],
...                     iris.domain.class_var,
...                     source=iris.domain)
>>> new_domain
[sepal length, petal length, color | iris]
```

classmethod **from_numpy**(*X*, *Y=None*, *metas=None*)

Create a domain corresponding to the given numpy arrays. This method is usually invoked from *Orange.data.Table.from_numpy()*.

All attributes are assumed to be continuous and are named "Feature <n>". Target variables are discrete if the only two values are 0 and 1; otherwise they are continuous. Discrete targets are named "Class <n>" and continuous are named "Target <n>". Domain is marked as *anonymous*, so data from any other domain of the same shape can be converted into this one and vice-versa.

Parameters

- **X** (*numpy.ndarray*) – 2-dimensional array with data
- **Y** (*numpy.ndarray* or None) – 1- of 2- dimensional data for target
- **metas** (*numpy.ndarray* or None) – meta attributes

Returns a new domain

Return type *Domain*

```
>>> import numpy as np
>>> from Orange.data import Domain
>>> X = np.arange(20, dtype=float).reshape(5, 4)
>>> Y = np.arange(5, dtype=int)
>>> domain = Domain.from_numpy(X, Y)
>>> domain
[Feature 1, Feature 2, Feature 3, Feature 4 | Class 1]
```

__getitem__(*idx*)

Return a variable descriptor from the given argument, which can be a descriptor, index or name. If *var* is a descriptor, the function returns this same object.

Parameters **idx** (int, str or *Variable*) – index, name or descriptor

Returns an instance of *Variable* described by *var*

Return type *Variable*

```
>>> iris.domain[1:3]
(ContinuousVariable('sepal width'), ContinuousVariable('petal length'))
```

__len__()
The number of variables (features and class attributes).

__contains__(*item*)
Return *True* if the item (*str*, *int*, *Variable*) is in the domain.

```
>>> "petal length" in iris.domain
True
>>> "age" in iris.domain
False
```

index(*var*)
Return the index of the given variable or meta attribute, represented with an instance of *Variable*, *int* or *str*.

```
>>> iris.domain.index("petal length")
2
```

has_discrete_attributes(*include_class=False*, *include_metas=False*)
Return *True* if domain has any discrete attributes. If *include_class* is set, the check includes the class attribute(s). If *include_metas* is set, the check includes the meta attributes.

```
>>> iris.domain.has_discrete_attributes()
False
>>> iris.domain.has_discrete_attributes(include_class=True)
True
```

has_continuous_attributes(*include_class=False*, *include_metas=False*)
Return *True* if domain has any continuous attributes. If *include_class* is set, the check includes the class attribute(s). If *include_metas* is set, the check includes the meta attributes.

```
>>> iris.domain.has_continuous_attributes()
True
```

Domain conversion

Domain descriptors also convert data instances between different domains.

In a typical scenario, we may want to discretize some continuous data before inducing a model. Discretizers (`Orange.preprocess`) construct a new data table with attribute descriptors (*Orange.data.variable*), that include the corresponding functions for conversion from continuous to discrete values. The trained model stores this domain descriptor and uses it to convert instances from the original domain to the discretized one at prediction phase.

In general, instances are converted between domains as follows.

- If the target attribute appears in the source domain, the value is copied; two attributes are considered the same if they have the same descriptor.
- If the target attribute descriptor defines a function for value transformation, the value is transformed.
- Otherwise, the value is marked as missing.

An exception to this rule are domains in which the anonymous flag is set. When the source or the target domain is anonymous, they match if they have the same number of variables and types. In this case, the data is copied without considering the attribute descriptors.

Variable Descriptors (`variable`)

Every variable is associated with a descriptor that stores its name and other properties. Descriptors serve three main purposes:

- conversion of values from textual format (e.g. when reading files) to the internal representation and back (e.g. when writing files or printing out);
- identification of variables: two variables from different data sets are considered to be the same if they have the same descriptor;
- conversion of values between domains or data sets, for instance from continuous to discrete data, using a pre-computed transformation.

Descriptors are most often constructed when loading the data from files.

```
>>> from Orange.data import Table
>>> iris = Table("iris")

>>> iris.domain.class_var
DiscreteVariable('iris')
>>> iris.domain.class_var.values
['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']

>>> iris.domain[0]
ContinuousVariable('sepal length')
>>> iris.domain[0].number_of_decimals
1
```

Some variables are derived from others. For instance, discretizing a continuous variable gives a new, discrete variable. The new variable can compute its values from the original one.

```
>>> from Orange.preprocess import DomainDiscretizer
>>> discretizer = DomainDiscretizer()
>>> d_iris = discretizer(iris)
>>> d_iris[0]
DiscreteVariable('D_sepal length')
>>> d_iris[0].values
['<5.2', '[5.2, 5.8)', '[5.8, 6.5)', '>=6.5']
```

See *Variable.compute_value* for a detailed explanation.

Constructors

Orange maintains lists of existing descriptors for variables. This facilitates the reuse of descriptors: if two data sets refer to the same variables, they should be assigned the same descriptors so that, for instance, a model trained on one data set can make predictions for the other.

Variable descriptors are seldom constructed in user scripts. When needed, this can be done by calling the constructor directly or by calling the class method *make*. The difference is that the latter returns an existing descriptor if there is one with the same name and which matches the other conditions, such as having the prescribed list of discrete values for *DiscreteVariable*:

```
>>> from Orange.data import ContinuousVariable
>>> age = ContinuousVariable.make("age")
>>> age1 = ContinuousVariable.make("age")
>>> age2 = ContinuousVariable("age")
>>> age is age1
True
>>> age is age2
False
```

The first line returns a new descriptor after not finding an existing desciptor for a continuous variable named "age". The second reuses the first descriptor. The last creates a new one since the constructor is invoked directly.

The distinction does not matter in most cases, but it is important when loading the data from different files. Orange uses the *make* constructor when loading data.

Base class

class Orange.data.**Variable**(*name=''*, *compute_value=None*)

The base class for variable descriptors contains the variable's name and some basic properties.

name

The name of the variable.

unknown_str

A set of values that represent unknowns in conversion from textual formats. Default is *{"?", ".", "", "NA", "~", None}*.

compute_value

A function for computing the variable's value when converting from another domain which does not contain this variable. The base class defines a static method *compute_value*, which returns *Unknown*. Non-primitive variables must redefine it to return *None*.

source_variable

An optional descriptor of the source variable - if any - from which this variable is derived and computed via *compute_value*.

attributes

A dictionary with user-defined attributes of the variable

master

The variable that this variable is a copy of. If a copy is made from a copy, the copy has a reference to the original master. If the variable is not a copy, it is its own master.

classmethod is_primitive()

True if the variable's values are stored as floats. Non-primitive variables can appear in the data only as meta attributes.

str_val(*val*)

Return a textual representation of variable's value *val*. Argument *val* must be a float (for primitive variables) or an arbitrary Python object (for non-primitives).

Derived classes must overload the function.

to_val(*s*)

Convert the given argument to a value of the variable. The argument can be a string, a number or *None*. For primitive variables, the base class provides a method that returns Unknown if *s* is found in *unknown_str*, and raises an exception otherwise. For non-primitive variables it returns the argument itself.

Derived classes of primitive variables must overload the function.

Parameters **s** (*str, float or None*) – value, represented as a number, string or *None*

Return type float or object

val_from_str_add(*s*)

Convert the given string to a value of the variable. The method is similar to *to_val* except that it only accepts strings and that it adds new values to the variable's domain where applicable.

The base class method calls *to_val*.

Parameters **s** (*str*) – symbolic representation of the value

Return type float or object

compute_value

Method *compute_value* is usually invoked behind the scenes in conversion of domains:

```
>>> from Orange.data import Table
>>> from Orange.preprocess import DomainDiscretizer

>>> iris = Table("iris")
>>> iris_1 = iris[::2]
>>> discretizer = DomainDiscretizer()
>>> d_iris_1 = discretizer(iris_1)

>>> d_iris_1[0]
DiscreteVariable('D_sepal length')
>>> d_iris_1[0].source_variable
ContinuousVariable('sepal length')
>>> d_iris_1[0].compute_value
<Orange.feature.discretization.Discretizer at 0x10d5108d0>
```

The data is loaded and the instances on even places are put into a new table, from which we compute discretized data. The discretized variable "D_sepal length" refers to the original as its source and stores a function for conversion of the original continuous values into the discrete. This function (and the corresponding functions for other variables) is used for converting the remaining data:

```
>>> iris_2 = iris[1::2]
>>> d_iris_2 = Table(d_iris_1.domain, iris_2)
>>> d_iris_2[0]
[<5.2, [2.8, 3), <1.6, <0.2 | Iris-setosa]
```

In the first line we select the instances with odd indices in the original table, that is, the data which was not used for computing the discretization. In the second line we construct a new data table with the discrete domain *d_iris_1.domain* and using the original data *iris_2*. Behind the scenes, the values for those variables in the destination domain (*d_iris_1.domain*) that do not appear in the source domain (*iris_2.domain*) are computed by passing the source data instance to the destination variables' *Variable.compute_value*.

This mechanism is used throughout Orange to compute all preprocessing on training data and applying the same transformations on the testing data without hassle.

Note that even such conversions are typically not coded in user scripts but implemented within the provided wrappers and cross-validation schemes.

Continuous variables

class Orange.data.**ContinuousVariable**(*name=''*, *number_of_decimals=None*, *compute_value=None*)

Descriptor for continuous variables.

number_of_decimals
The number of decimals when the value is printed out (default: 3).

adjust_decimals
A flag regulating whether the *number_of_decimals* is being adjusted by `to_val`.

The value of *number_of_decimals* is set to 3 and *adjust_decimals* is set to 2. When `val_from_str_add` is called for the first time with a string as an argument, *number_of_decimals* is set to the number of decimals in the string and *adjust_decimals* is set to 1. In the subsequent calls of *to_val*, the nubmer of decimals is increased if the string argument has a larger number of decimals.

If the *number_of_decimals* is set manually, *adjust_decimals* is set to 0 to prevent changes by *to_val*.

make(*name*)
Return an existing continuous variable with the given name, or construct and return a new one.

is_primitive()
True if the variable's values are stored as floats. Non-primitive variables can appear in the data only as meta attributes.

str_val(*val*)
Return the value as a string with the prescribed number of decimals.

to_val(*s*)
Convert a value, given as an instance of an arbitrary type, to a float.

val_from_str_add(*s*)
Convert a value from a string and adjust the number of decimals if *adjust_decimals* is non-zero.

Discrete variables

class Orange.data.**DiscreteVariable**(*name=''*, *values=()*, *ordered=False*, *base_value=-1*, *compute_value=None*)

Descriptor for symbolic, discrete variables. Values of discrete variables are stored as floats; the numbers corresponds to indices in the list of values.

values
A list of variable's values.

ordered
Some algorithms (and, in particular, visualizations) may sometime reorder the values of the variable, e.g. alphabetically. This flag hints that the given order of values is "natural" (e.g. "small", "middle", "large") and should not be changed.

base_value
The index of the base value, or -1 if there is none. The base value is used in some methods like, for instance, when creating dummy variables for regression.

classmethod make(*name*, *values=()*, *ordered=False*, *base_value=-1*)
Return a variable with the given name and other properties. The method first looks for a compatible existing variable: the existing variable must have the same name and both variables must have either ordered or unordered values. If values are ordered, the order must be compatible: all common values must have the same order. If values are unordered, the existing variable must have at least one common value with the new one, except when any of the two lists of values is empty.

If a compatible variable is find, it is returned, with missing values appended to the end of the list. If there is no explicit order, the values are ordered using `ordered_values`. Otherwise, it constructs and returns a new variable descriptor.

Parameters

- **name** (*str*) – the name of the variable
- **values** (`list`) – symbolic values for the variable
- **ordered** (*bool*) – tells whether the order of values is fixed
- **base_value** (*int*) – the index of the base value, or -1 if there is none

Returns an existing compatible variable or *None*

is_primitive()
True if the variable's values are stored as floats. Non-primitive variables can appear in the data only as meta attributes.

str_val(*val*)
Return a textual representation of the value (*self.values[int(val)]*) or "?" if the value is unknown.

Parameters **val** (*float (should be whole number)*) – value

Return type str

to_val(*s*)
Convert the given argument to a value of the variable (*float*). If the argument is numeric, its value is returned without checking whether it is integer and within bounds. *Unknown* is returned if the argument is one of the representations for unknown values. Otherwise, the argument must be a string and the method returns its index in *values*.

Parameters **s** – values, represented as a number, string or *None*

Return type float

val_from_str_add(*s*)
Similar to *to_val*, except that it accepts only strings and that it adds the value to the list if it does not exist yet.

Parameters **s** (*str*) – symbolic representation of the value

Return type float

String variables

class Orange.data.**StringVariable**(*name=''*, *compute_value=None*)
Descriptor for string variables. String variables can only appear as meta attributes.

make(*name*)
Return an existing continuous variable with the given name, or construct and return a new one.

is_primitive()
True if the variable's values are stored as floats. Non-primitive variables can appear in the data only as meta attributes.

static str_val(*val*)
Return a string representation of the value.

to_val(*s*)
Return the value as a string. If it is already a string, the same object is returned.

val_from_str_add(*s*)
Return the value as a string. If it is already a string, the same object is returned.

Time variables

Time variables are continuous variables with value 0 on the Unix epoch, 1 January 1970 00:00:00.0 UTC. Positive numbers are dates beyond this date, and negative dates before. Due to limitation of Python `datetime` module, only dates in 1 A.D. or later are supported.

class `Orange.data.`**TimeVariable**(**args*, ***kwargs*)
TimeVariable is a continuous variable with Unix epoch (1970-01-01 00:00:00+0000) as the origin (0.0). Later dates are positive real numbers (equivalent to Unix timestamp, with microseconds in the fraction part), and the dates before it map to the negative real numbers.

Unfortunately due to limitation of Python datetime, only dates with year >= 1 (A.D.) are supported.

If time is specified without a date, Unix epoch is assumed.

If time is specified wihout an UTC offset, localtime is assumed.

parse(*datestr*)
Return *datestr*, a datetime provided in one of ISO 8601 formats, parsed as a real number. Value 0 marks the Unix epoch, positive values are the dates after it, negative before.

If date is unspecified, epoch date is assumed.

If time is unspecified, 00:00:00.0 is assumed.

If timezone is unspecified, local time is assumed.

Values (`value`)

class `Orange.data.variable.`**Value**(*_*, *__=nan*)
The class representing a value. The class is not used to store values but only to return them in contexts in which we want the value to be accompanied with the descriptor, for instance to print the symbolic value of discrete variables.

The class is derived from *float*, with an additional attribute *variable* which holds the descriptor of type *Orange.data.Variable*. If the value continuous or discrete, it is stored as a float. Other types of values, like strings, are stored in the attribute *value*.

The class overloads the methods for printing out the value: *variable.repr_val* and *variable.str_val* are used to get a suitable representation of the value.

Equivalence operator is overloaded as follows:

- unknown values are equal; if one value is unknown and the other is not, they are different;
- if the value is compared with the string, the value is converted to a string using *variable.str_val* and the two strings are compared
- if the value is stored in attribute *value*, it is compared with the given other value
- otherwise, the inherited comparison operator for *float* is called.

Finally, value defines a hash, so values can be put in sets and appear as keys in dictionaries.

variable(*:obj:'Orange.data.Variable'*)
Descriptor; used for printing out and for comparing with strings

value
Value; the value can be of arbitrary type and is used only for variables that are neither discrete nor continuous. If *value* is *None*, the derived *float* value is used.

Data Instance (`instance`)

Class *Instance* represents a data instance, typically retrieved from a *Orange.data.Table* or `Orange.data.sql.SqlTable`. The base class contains a copy of the data; modifying does not change the data in the storage from which the instance was retrieved. Derived classes (e.g. `Orange.data.table.RowInstance`) can represent views into various data storages, therefore changing them actually changes the data.

Like data tables, every data instance is associated with a domain and its data is split into attributes, classes, meta attributes and the weight. Its constructor thus requires a domain and, optionally, data. For the following example, we borrow the domain from the Iris data set.

```
>>> from Orange.data import Table, Instance
>>> iris = Table("iris")
>>> inst = Instance(iris.domain, [5.2, 3.8, 1.4, 0.5, "Iris-virginica"])
>>> inst
[5.2, 3.8, 1.4, 0.5 | Iris-virginica]
>>> inst0 = Instance(iris.domain)
>>> inst0
[?, ?, ?, ? | ?]
```

The instance's data can be retrieved through attributes `x`, `y` and `metas`.

```
>>> inst.x
array([ 5.2,  3.8,  1.4,  0.5])
>>> inst.y
array([ 2.])
>>> inst.metas
array([], dtype=object)
```

Other utility functions provide for easier access to the instances data.

```
>>> inst.get_class()
Value('iris', Iris-virginica)
>>> for e in inst.attributes():
...     print(e)
...
5.2
3.8
1.4
0.5
```

class `Orange.data.`**`Instance`**(*domain*, *data=None*, *id=None*)
Constructor requires a domain and the data as numpy array, an existing instance from the same or another domain or any Python iterable.

Domain can be omitted it the data is given as an existing data instances.

When the instance is not from the given domain, Orange converts it.

```
>>> from Orange.preprocess import DomainDiscretizer
>>> discretizer = DomainDiscretizer()
>>> d_iris = discretizer(iris)
>>> d_inst = Instance(d_iris, inst)
```

domain
The domain describing the instance's values.

x
Instance's attributes as a 1-dimensional numpy array whose length equals *len(self.domain.attributes)*.

y
Instance's classes as a 1-dimensional numpy array whose length equals *len(self.domain.attributes)*.

metas
Instance's meta attributes as a 1-dimensional numpy array whose length equals *len(self.domain.attributes)*.

list
All instance's values, including attributes, classes and meta attributes, as a list whose length equals *len(self.domain.attributes) + len(self.domain.class_vars) + len(self.domain.metas)*.

weight
The weight of the data instance. Default is 1.

attributes()
Return iterator over the instance's attributes

classes()
Return iterator over the instance's class attributes

get_class()
Return the class value as an instance of `Orange.data.Value`. Throws an exception if there are multiple classes.

get_classes()
Return the class value as a list of instances of `Orange.data.Value`.

set_class(*value*)
Set the instance's class. Throws an exception if there are multiple classes.

Rows of Data Tables

class `Orange.data.`**RowInstance**(*table*, *row_index*)
RowInstance is a specialization of *Instance* that represents a row of *Orange.data.Table*. *RowInstance* is returned by indexing a *Table*.

The difference between *Instance* and *RowInstance* is that the latter represents a view into the table: changing the *RowInstance* changes the data in the table:

```
>>> iris[42]
[4.4, 3.2, 1.3, 0.2 | Iris-setosa]
>>> inst = iris[42]
>>> inst.set_class("Iris-virginica")
>>> iris[42]
[4.4, 3.2, 1.3, 0.2 | Iris-virginica]
```

Dense tables can also be modified directly through `x`, `y` and `metas`.

```
>>> inst.x[0] = 5
>>> iris[42]
[5.0, 3.2, 1.3, 0.2 | Iris-virginica]
```

Sparse tables cannot be changed in this way.

Data Filters (`filter`)

Instances of classes derived from *Filter* are used for filtering the data.

When called with an individual data instance (*Orange.data.Instance*), they accept or reject the instance by returning either *True* or *False*.

When called with a data storage (e.g. an instance of *Orange.data.Table*) they check whether the corresponding class provides the method that implements the particular filter. If so, the method is called and the result should be of the same type as the storage; e.g., filter methods of *Orange.data.Table* return new instances of *Orange.data.Table*, and filter methods of SQL proxies return new SQL proxies.

If the class corresponding to the storage does not implement a particular filter, the fallback computes the indices of the rows to be selected and returns *data[indices]*.

class Orange.data.filter.**Filter**(*negate=False*)
The base class for filters.

negate
Reverts the selection

class Orange.data.filter.**IsDefined**(*columns=None*, *negate=False*)
Select the data instances with no undefined values. The check can be restricted to a subset of columns.

The filter's behaviour may depend upon the storage implementation.

In particular, *Table* with sparse matrix representation will select all data instances whose values are defined, even if they are zero. However, if individual columns are checked, it will select all rows with non-zero entries for this columns, disregarding whether they are stored as zero or omitted.

columns
The columns to be checked, given as a sequence of indices, names or *Orange.data.Variable*.

class Orange.data.filter.**HasClass**(*negate=False*)
Return all rows for which the class value is known.

Orange.data.Table implements the filter on the sparse data so that it returns all rows for which all class values are defined, even if they equal zero.

class Orange.data.filter.**Random**(*prob=None*, *negate=False*)
Return a random selection of data instances.

prob
The proportion (if below 1) or the probability (if 1 or above) of selected instances

class Orange.data.filter.**SameValue**(*column*, *value*, *negate=False*)
Return the data instances with the given value in the specified column.

column
The column, described by an index, a string or *Orange.data.Variable*.

value
The reference value

class Orange.data.filter.**Values**(*conditions*, *conjunction=True*, *negate=False*)
Select the data instances based on conjunction or disjunction of filters derived from *ValueFilter* that check values of individual features or another (nested) Values filter.

conditions
A list of conditions, derived from *ValueFilter* or *Values*

conjunction
If *True*, the filter computes a conjunction, otherwise a disjunction

negate
Revert the selection

class Orange.data.filter.**ValueFilter**(*column*)
The base class for subfilters that check individual values of data instances. Derived classes handle discrete, continuous and string attributes. These filters are used to compose conditions in *Orange.data.filter.Values*.

The internal implementation of *filter.Values* in data storages, like *Orange.data.Table*, recognize these filters and retrieve their, attributes, like operators and reference values, but do not call them.

The fallback implementation of *Orange.data.filter.Values* calls the subfilters with individual data instances, which is very inefficient.

column
The column to which the filter applies (int, str or *Orange.data.Variable*).

class Orange.data.filter.**FilterDiscrete**(*column*, *values*)
Subfilter for discrete variables, which selects the instances whose value matches one of the given values.

column
The column to which the filter applies (int, str or *Orange.data.Variable*).

values
The list (or a set) of accepted values. If None, it checks whether the value is defined.

class Orange.data.filter.**FilterContinuous**(*position*, *oper*, *ref=None*, *max=None*, *min=None*)
Subfilter for continuous variables.

column
The column to which the filter applies (int, str or *Orange.data.Variable*).

ref
The reference value; also aliased to *min* for operators *Between* and *Outside*.

max
The upper threshold for operators *Between* and *Outside*.

oper
The operator; should be *FilterContinuous.Equal*, *NotEqual*, *Less*, *LessEqual*, *Greater*, *GreaterEqual*, *Between*, *Outside* or *IsDefined*.

Type
alias of *FilterContinuous*

class Orange.data.filter.**FilterString**(*position*, *oper*, *ref=None*, *max=None*, *case_sensitive=True*, ***a*)
Subfilter for string variables.

column
The column to which the filter applies (int, str or *Orange.data.Variable*).

ref
The reference value; also aliased to *min* for operators *Between* and *Outside*.

max
The upper threshold for operators *Between* and *Outside*.

oper
The operator; should be *FilterString.Equal*, *NotEqual*, *Less*, *LessEqual*, *Greater*, *GreaterEqual*, *Between*, *Outside*, *Contains*, *StartsWith*, *EndsWith* or *IsDefined*.

case_sensitive
Tells whether the comparisons are case sensitive

Type
alias of *FilterString*

class Orange.data.filter.**FilterStringList**(*column*, *values*, *case_sensitive=True*)
Subfilter for strings variables which checks whether the value is in the given list of accepted values.

column
The column to which the filter applies (int, str or *Orange.data.Variable*).

values
The list (or a set) of accepted values.

case_sensitive
Tells whether the comparisons are case sensitive

class Orange.data.filter.**FilterRegex**(*column*, *pattern*, *flags=0*)
Filter that checks whether the values match the regular expression.

Loading and saving data (io)

Orange.data.Table supports loading from several file formats:

- Comma-separated values (*.csv) file,
- Tab-separated values (*.tab, *.tsv) file,
- Excel spreadsheet (*.xls, *.xlsx),
- Basket file,
- Python pickle.

In addition, the text-based files (CSV, TSV) can be compressed with gzip, bzip2 or xz (e.g. *.csv.gz).

Header Format

The data in CSV, TSV, and Excel files can be described in an extended three-line header format, or a condensed single-line header format.

Three-line header format

A three-line header consists of:

1. **Feature names** on the first line. Feature names can include any combination of characters.
2. **Feature types** on the second line. The type is determined automatically, or, if set, can be any of the following:
   - `discrete` (or `d`) — imported as *Orange.data.DiscreteVariable*,
   - a space-separated **list of discrete values**, like "`male female`", which will result in *Orange.data.DiscreteVariable* with those values and in that order. If the individual values contain a space character, it needs to be escaped (prefixed) with, as common, a backslash ('\') character.
   - `continuous` (or `c`) — imported as *Orange.data.ContinuousVariable*,
   - `string` (or `s`, or `text`) — imported as *Orange.data.StringVariable*,
   - `time` (or `t`) — imported as *Orange.data.TimeVariable*, if the values parse as ISO 8601 date/time formats,
   - `basket` — used for storing sparse data. More on basket formats in a dedicated section.

3. **Flags** (optional) on the third header line. Feature's flag can be empty, or it can contain, space-separated, a consistent combination of:
    - `class` (or `c`) — feature will be imported as a class variable. Most algorithms expect a single class variable.
    - `meta` (or `m`) — feature will be imported as a meta-attribute, just describing the data instance but not actually used for learning,
    - `weight` (or `w`) — the feature marks the weight of examples (in algorithms that support weighted examples),
    - `ignore` (or `i`) — feature will not be imported,
    - `<key>=<value>` custom attributes.

Example of iris dataset in Orange's three-line format (`iris.tab`).

```
sepal length        sepal width        petal length        petal width        iris
c        c        c        c        d
                                    class
5.1        3.5        1.4        0.2        Iris-setosa
4.9        3.0        1.4        0.2        Iris-setosa
4.7        3.2        1.3        0.2        Iris-setosa
4.6        3.1        1.5        0.2        Iris-setosa
```

Single-line header format

Single-line header consists of feature names prefixed by an optional "`<flags>#`" string, i.e. flags followed by a hash ('#') sign. The flags can be a consistent combination of:

- `c` for class feature,
- `i` for feature to be ignored,
- `m` for meta attributes (not used in learning),
- `C` for features that are continuous,
- `D` for features that are discrete,
- `T` for features that represent date and/or time in one of the ISO 8601 formats,
- `S` for string features.

If some (all) names or flags are omitted, the names, types, and flags are discerned automatically, and correctly (most of the time).

Baskets

Baskets can be used for storing sparse data in tab delimited files. They were specifically designed for text mining needs. If text mining and sparse data is not your business, you can skip this section.

Baskets are given as a list of space-separated `<name>=<value>` atoms. A continuous meta attribute named `<name>` will be created and added to the domain as optional if it is not already there. A meta value for that variable will be added to the example. If the value is 1, you can omit the `=<value>` part.

It is not possible to put meta attributes of other types than continuous in the basket.

A tab delimited file with a basket can look like this:

```
K       Ca      b_foo     Ba  y
c       c       basket    c   c
        meta              i   class
0.06    8.75    a b a c   0   1
0.48            b=2 d     0   1
0.39    7.78              0   1
0.57    8.22    c=13      0   1
```

These are the examples read from such a file:

```
[0.06, 1], {"Ca":8.75, "a":2.000, "b":1.000, "c":1.000}
[0.48, 1], {"Ca":?, "b":2.000, "d":1.000}
[0.39, 1], {"Ca":7.78}
[0.57, 1], {"Ca":8.22, "c":13.000}
```

It is recommended to have the basket as the last column, especially if it contains a lot of data.

Note a few things. The basket column's name, `b_foo`, is not used. In the first example, the value of `a` is 2 since it appears twice. The ordinary meta attribute, `Ca`, appears in all examples, even in those where its value is undefined. Meta attributes from the basket appear only where they are defined. This is due to the different nature of these meta attributes: `Ca` is required while the others are optional.

```
>>> d.domain.metas()
{-6: FloatVariable 'd', -22: FloatVariable 'Ca', -5: FloatVariable 'c', -4:␣
↪FloatVariable 'b', -3: FloatVariable 'a'}
```

To fully understand all this, you should read the documentation on meta attributes in Domain and on the *basket file format* (a simple format that is limited to baskets only).

Basket Format

Basket files (.basket) are suitable for representing sparse data. Each example is represented by a line in the file. The line is written as a comma-separated list of name-value pairs. Here's an example of such file.

```
nobody, expects, the, Spanish, Inquisition=5
our, chief, weapon, is, surprise=3, surprise=2, and, fear,fear, and, surprise
our, two, weapons, are, fear, and, surprise, and, ruthless, efficiency
to, the, Pope, and, nice, red, uniforms, oh damn
```

The file contains four examples. The first examples has five attributes defined, "nobody", "expects", "the", "Spanish" and "Inquisition"; the first four have (the default) value of 1.0 and the last has a value of 5.0.

The attributes that appear in the domain aren't defined in any headers or even separate files, as with other formats supported by Orange.

If attribute appears more than once, its values are added. For instance, the value of attribute "surprise" in the second examples is 6.0 and the value of "fear" is 2.0; the former appears three times with values of 3.0, 2.0 and 1.0, and the latter appears twice with value of 1.0.

All attributes are loaded as optional meta-attributes, so zero values don't take any memory (unless they are given, but initialized to zero). See also section on meta attributes in the reference for domain descriptors.

Notice that at the time of writing this reference only association rules can directly use examples presented in the basket format.

Data Preprocessing (preprocess)

Preprocessing module contains data processing utilities like data discretization, continuization, imputation and transformation.

Impute

Imputation replaces missing values with new values (or omits such features).

```
from Orange.data import Table
from Orange.preprocess import Impute

data = Table("heart-disease.tab")
imputer = Impute()

impute_heart = imputer(data)
```

There are several imputation methods one can use.

```
from Orange.data import Table
from Orange.preprocess import Impute, Average

data = Table("heart_disease.tab")
imputer = Impute(method=Average())
impute_heart = imputer(data)
```

Discretization

Discretization replaces continuous features with the corresponding categorical features:

```
import Orange
iris = Orange.data.Table("iris.tab")
disc = Orange.preprocess.Discretize()
disc.method = Orange.preprocess.discretize.EqualFreq(n=3)
d_iris = disc(iris)

print("Original data set:")
for e in iris[:3]:
    print(e)

print("Discretized data set:")
for e in d_iris[:3]:
    print(e)
```

The variable in the new data table indicate the bins to which the original values belong.

```
Original data set:
[5.1, 3.5, 1.4, 0.2 | Iris-setosa]
[4.9, 3.0, 1.4, 0.2 | Iris-setosa]
[4.7, 3.2, 1.3, 0.2 | Iris-setosa]
Discretized data set:
[<5.5, >=3.2, <2.5, <0.8 | Iris-setosa]
[<5.5, [2.8, 3.2), <2.5, <0.8 | Iris-setosa]
[<5.5, >=3.2, <2.5, <0.8 | Iris-setosa]
```

Default discretization method (four bins with approximatelly equal number of data instances) can be replaced with other methods.

```
disc = Orange.preprocess.Discretize()
disc.method = Orange.preprocess.discretize.EqualFreq(n=2)
d_disc_iris = disc(iris)
```

Discretization Algorithms

class Orange.preprocess.discretize.**EqualWidth**(*n=4*)
Discretization into a fixed number of bins with equal widths.

n
Number of bins (default: 4).

class Orange.preprocess.discretize.**EqualFreq**(*n=4*)
Discretization into bins with approximately equal number of data instances.

n
Number of bins (default: 4). The actual number may be lower if the variable has less than n distinct values.

class Orange.preprocess.discretize.**EntropyMDL**(*force=False*)
Discretization into bins inferred by recursively splitting the values to minimize the class-entropy. The procedure stops when further splits would decrease the entropy for less than the corresponding increase of minimal description length (MDL). [FayyadIrani93].

If there are no suitable cut-off points, the procedure returns a single bin, which means that the new feature is constant and can be removed.

force
Induce at least one cut-off point, even when its information gain is lower than MDL (default: False).

To add a new discretization, derive it from Discretization.

class Orange.preprocess.discretize.**Discretization**
Abstract base class for discretization classes.

Continuization

class Orange.preprocess.**Continuize**
Given a data table, return a new table in which the discretize attributes are replaced with continuous or removed.

- binary variables are transformed into 0.0/1.0 or -1.0/1.0 indicator variables, depending upon the argument zero_based.
- multinomial variables are treated according to the argument multinomial_treatment.
- discrete attribute with only one possible value are removed;

```
import Orange
titanic = Orange.data.Table("titanic")
continuizer = Orange.preprocess.Continuize()
titanic1 = continuizer(titanic)
```

The class has a number of attributes that can be set either in constructor or, later, as attributes.

zero_based
Determines the value used as the "low" value of the variable. When binary variables are transformed into continuous or when multivalued variable is transformed into multiple variables, the transformed variable can either have values 0.0 and 1.0 (default, `zero_based=True`) or -1.0 and 1.0 (`zero_based=False`).

multinomial_treatment
Defines the treatment of multinomial variables.

`Continuize.Indicators`

The variable is replaced by indicator variables, each corresponding to one value of the original variable. For each value of the original attribute, only the corresponding new attribute will have a value of one and others will be zero. This is the default behaviour.

Note that these variables are not independent, so they cannot be used (directly) in, for instance, linear or logistic regression.

For example, data set "titanic" has feature "status" with values "crew", "first", "second" and "third", in that order. Its value for the 15th row is "first". Continuization replaces the variable with variables "status=crew", "status=first", "status=second" and "status=third". After

```
continuizer = Orange.preprocess.Continuize()
titanic1 = continuizer(titanic)
```

we have

```
>>> titanic.domain
[status, age, sex | survived]
>>> titanic1.domain
[status=crew, status=first, status=second, status=third,
 age=adult, age=child, sex=female, sex=male | survived]
```

For the 15th row, the variable "status=first" has value 1 and the values of the other three variables are 0:

```
>>> print(titanic[15])
[first, adult, male | yes]
>>> print(titanic1[15])
[0.000, 1.000, 0.000, 0.000, 1.000, 0.000, 0.000, 1.000 | yes]
```

Continuize.FirstAsBase Similar to the above, except that it creates indicators for all values except the first one, according to the order in the variable's *values* attribute. If all indicators in the transformed data instance are 0, the original instance had the first value of the corresponding variable.

If the variable descriptor defines the *base_value*, the specified value is used as base instead of the first one.

Continuizing the variable "status" with this setting gives variables "status=first", "status=second" and "status=third". If all of them were 0, the status of the original data instance was "crew".

```
>>> continuizer.multinomial_treatment = continuizer.FirstAsBase
>>> continuizer(titanic).domain
[status=first, status=second, status=third, age=child, sex=male |␣
↪survived]
```

Continuize.FrequentAsBase Like above, except that the most frequent value is used as the base. If there are multiple most frequent values, the one with the lowest index in *values* is used. The frequency of values is extracted from data, so this option does not work if only the domain is given.

Continuizing the Titanic data in this way differs from the above by the attributes sex: instead of "sex=male" it constructs "sex=female" since there were more females than males on Titanic.

```
>>> continuizer.multinomial_treatment = continuizer.FrequentAsBase
>>> continuizer(titanic).domain
[status=first, status=second, status=third, age=child, sex=female |␣
↪survived]
```

Continuize.Remove Discrete variables are removed.

```
>>> continuizer.multinomial_treatment = continuizer.Remove
>>> continuizer(titanic).domain
[ | survived]
```

Continuize.RemoveMultinomial Discrete variables with more than two values are removed. Binary variables are treated the same as in *FirstAsBase*.

```
>>> continuizer.multinomial_treatment = continuizer.RemoveMultinomial
>>> continuizer(titanic).domain
[age=child, sex=male | survived]
```

Continuize.ReportError Raise an error if there are any multinomial variables in the data.

Continuize.AsOrdinal Multinomial variables are treated as ordinal and replaced by continuous variables with indices within *values*, e.g. 0, 1, 2, 3...

```
>>> continuizer.multinomial_treatment = continuizer.AsOrdinal
>>> titanic1 = continuizer(titanic)
>>> titanic[700]
[third, adult, male | no]
>>> titanic1[700]
[3.000, 0.000, 1.000 | no]
```

Continuize.AsNormalizedOrdinal As above, except that the resulting continuous value will be from range 0 to 1, e.g. 0, 0.333, 0.667, 1 for a four-valued variable:

```
>>> continuizer.multinomial_treatment = continuizer.AsNormalizedOrdinal
>>> titanic1 = continuizer(titanic)
>>> titanic1[700]
[1.000, 0.000, 1.000 | no]
>>> titanic1[15]
[0.333, 0.000, 1.000 | yes]
```

transform_class
If `True` the class is replaced by continuous attributes or normalized as well. Multiclass problems are thus transformed to multitarget ones. (Default: `False`)

class `Orange.preprocess.`**DomainContinuizer**
Construct a domain in which discrete attributes are replaced by continuous.

```
domain_continuizer = Orange.preprocess.DomainContinuizer()
domain1 = domain_continuizer(titanic)
```

Orange.preprocess.Continuize calls *DomainContinuizer* to construct the domain.

Domain continuizers can be given either a data set or a domain, and return a new domain. When given only the domain, use the most frequent value as the base value.

By default, the class does not change continuous and class attributes, discrete attributes are replaced with N attributes (`Indicators`) with values 0 and 1.

Normalization

class `Orange.preprocess.`**`Normalize`**(*zero_based=True*, *norm_type=Normalize.NormalizeBySD*, *transform_class=False*)

Construct a preprocessor for normalization of features. Given a data table, preprocessor returns a new table in which the continuous attributes are normalized.

Parameters **zero_based** : bool (default=True)

Determines the value used as the "low" value of the variable. It determines the interval for normalized continuous variables (either [-1, 1] or [0, 1]).

norm_type : NormTypes (default: Normalize.NormalizeBySD)

Normalization type. If Normalize.NormalizeBySD, the values are replaced with standardized values by subtracting the average value and dividing by the standard deviation. Attribute zero_based has no effect on this standardization.

If Normalize.NormalizeBySpan, the values are replaced with normalized values by subtracting min value of the data and dividing by span (max - min).

transform_class : bool (default=False)

If True the class is normalized as well.

Examples

```
>>> from Orange.data import Table
>>> from Orange.preprocess import Normalize
>>> data = Table("iris")
>>> normalizer = Normalize(norm_type=Normalize.NormalizeBySpan)
>>> normalized_data = normalizer(data)
```

Randomization

class `Orange.preprocess.`**`Randomize`**(*rand_type=Randomize.RandomizeAttributes*, *rand_seed=None*)

Construct a preprocessor for randomization of classes, attributes and/or metas. Given a data table, preprocessor returns a new table in which the data is shuffled.

Parameters **rand_type** : RandTypes (default: Randomize.RandomizeClasses)

Randomization type. If Randomize.RandomizeClasses, classes are shuffled. If Randomize.RandomizeAttributes, attributes are shuffled. If Randomize.RandomizeMetas, metas are shuffled.

rand_seed : int (optional)

Random seed

Examples

```
>>> from Orange.data import Table
>>> from Orange.preprocess import Randomize
>>> data = Table("iris")
>>> randomizer = Randomize(Randomize.RandomizeClasses)
>>> randomized_data = randomizer(data)
```

Remove

class Orange.preprocess.**Remove** (*attr_flags=0*, *class_flags=0*, *meta_flags=0*)

Construct a preprocessor for removing constant features/classes and unused values. Given a data table, preprocessor returns a new table and a list of results. In the new table, the constant features/classes and unused values are removed. The list of results consists of two dictionaries. The first one contains numbers of 'removed', 'reduced' and 'sorted' features. The second one contains numbers of 'removed', 'reduced' and 'sorted' features.

Parameters **attr_flags** : int (default: 0)

If SortValues, values of discrete attributes are sorted. If RemoveConstant, unused attributes are removed. If RemoveUnusedValues, unused values are removed from discrete attributes. It is possible to merge operations in one by summing several types.

class_flags: int (default: 0)

If SortValues, values of discrete class attributes are sorted. If RemoveConstant, unused class attributes are removed. If RemoveUnusedValues, unused values are removed from discrete class attributes. It is possible to merge operations in one by summing several types.

Examples

```
>>> from Orange.data import Table
>>> from Orange.preprocess import Remove
>>> data = Table("zoo")[:10]
>>> flags = sum([Remove.SortValues, Remove.RemoveConstant, Remove.
↪RemoveUnusedValues])
>>> remover = Remove(attr_flags=flags, class_flags=flags)
>>> new_data = remover(data)
>>> attr_results, class_results = remover.attr_results, remover.class_results
```

Feature selection

Feature scoring

Feature scoring is an assessment of the usefulness of features for prediction of the dependant (class) variable. Orange provides classes that compute the common feature scores for classification and regression.

The code below computes the information gain of feature "tear_rate" in the Lenses data set:

```
>>> data = Orange.data.Table("lenses")
>>> Orange.preprocess.score.InfoGain(data, "tear_rate")
0.54879494069539858
```

An alternative way of invoking the scorers is to construct the scoring object and calculate the scores for all the features at once, like in the following example:

```
>>> gain = Orange.preprocess.score.InfoGain()
>>> scores = gain(data)
>>> for attr, score in zip(data.domain.attributes, scores):
...     print('%.3f' % score, attr.name)
0.039 age
0.040 prescription
0.377 astigmatic
0.549 tear_rate
```

Feature scoring methods work on different feature types (continuous or discrete) and different types of target variables (i.e. in classification or regression problems). Refer to method's *feature_type* and *class_type* attributes for intended type or employ preprocessing methods (e.g. discretization) for conversion between data types.

class Orange.preprocess.score.**ANOVA**

A wrapper for *sklearn.feature_selection.univariate_selection.f_classif*. The following is the documentation from scikit-learn.

Compute the ANOVA F-value for the provided sample.

Read more in the User Guide.

feature_type
alias of ContinuousVariable

class_type
alias of DiscreteVariable

class Orange.preprocess.score.**Chi2**

A wrapper for *sklearn.feature_selection.univariate_selection.chi2*. The following is the documentation from scikit-learn.

Compute chi-squared stats between each non-negative feature and class.

This score can be used to select the n_features features with the highest values for the test chi-squared statistic from X, which must contain only non-negative features such as booleans or frequencies (e.g., term counts in document classification), relative to the classes.

Recall that the chi-square test measures dependence between stochastic variables, so using this function "weeds out" the features that are the most likely to be independent of class and therefore irrelevant for classification.

Read more in the User Guide.

feature_type
alias of DiscreteVariable

class_type
alias of DiscreteVariable

class Orange.preprocess.score.**GainRatio**
Information gain ratio is the ratio between information gain and the entropy of the feature's value distribution. The score was introduced in *[Quinlan198689]* to alleviate overestimation for multi-valued features. See Wikipedia entry on gain ratio.

class_type
alias of DiscreteVariable

feature_type
alias of DiscreteVariable

class Orange.preprocess.score.**Gini**
Gini impurity is the probability that two randomly chosen instances will have different classes. See Wikipedia entry on Gini impurity.

class_type
alias of DiscreteVariable

feature_type
alias of DiscreteVariable

class Orange.preprocess.score.**InfoGain**
Information gain is the expected decrease of entropy. See Wikipedia entry on information gain.

class_type
alias of DiscreteVariable

feature_type
alias of DiscreteVariable

class Orange.preprocess.score.**UnivariateLinearRegression**

A wrapper for *sklearn.feature_selection.univariate_selection.f_regression*. The following is the documentation from scikit-learn.

Univariate linear regression tests.

Linear model for testing the individual effect of each of many regressors. This is a scoring function to be used in a feature seletion procedure, not a free standing feature selection procedure.

This is done in 2 steps:

1. The correlation between each regressor and the target is computed, that is, ((X[:, i] - mean(X[:, i])) * (y - mean_y)) / (std(X[:, i]) * std(y)).
2. It is converted to an F score then to a p-value.

For more on usage see the User Guide.

feature_type
alias of ContinuousVariable

class_type
alias of ContinuousVariable

class Orange.preprocess.score.**FCBF**
Fast Correlation-Based Filter. Described in:

Yu, L., Liu, H., Feature selection for high-dimensional data: A fast correlation-based filter solution. 2003. http://www.aaai.org/Papers/ICML/2003/ICML03-111.pdf

class_type
alias of DiscreteVariable

feature_type
alias of DiscreteVariable

class Orange.preprocess.score.**ReliefF**(*n_iterations=50*, *k_nearest=10*)
ReliefF algorithm. Contrary to most other scorers, Relief family of algorithms is not as myoptic but tends to give unreliable results with datasets with lots (hundreds) of features.

Robnik-Šikonja, M., Kononenko, I. Theoretical and empirical analysis of ReliefF and RReliefF. 2003. http://lkm.fri.uni-lj.si/rmarko/papers/robnik03-mlj.pdf

feature_type
alias of Variable

class_type
alias of DiscreteVariable

class Orange.preprocess.score.**RReliefF**(*n_iterations=50*, *k_nearest=50*)

feature_type
alias of Variable

class_type
alias of ContinuousVariable

Additionally, you can use the score_data() method of some learners (Orange.classification.LinearRegressionLearner, Orange.regression.LogisticRegressionLearner, *Orange.classification.RandomForestLearner*, and *Orange.regression.RandomForestRegressionLearner*) to obtain the feature scores as calculated by these learners. For example:

```
>>> learner = Orange.classification.LogisticRegressionLearner()
>>> learner.score_data(data)
[0.31571299907366146,
 0.28286199971877485,
 0.67496525667835794,
 0.99930286901257692]
```

Feature selection

We can use feature selection to limit the analysis to only the most relevant or informative features in the data set.

Feature selection with a scoring method that works on continuous features will retain all discrete features and vice versa.

The code below constructs a new data set consisting of two best features according to the ANOVA method:

```
>>> data = Orange.data.Table("wine")
>>> anova = Orange.preprocess.score.ANOVA()
>>> selector = Orange.preprocess.SelectBestFeatures(method=anova, k=2)
>>> data2 = selector(data)
>>> data2.domain
[Flavanoids, Proline | Wine]
```

class Orange.preprocess.**SelectBestFeatures**(*method=None*, *k=None*, *threshold=None*, *decreasing=True*)

A feature selector that builds a new data set consisting of either the top *k* features or all those that exceed a given *threshold*. Features are scored using the provided feature scoring *method*. By default it is assumed that feature importance diminishes with decreasing scores.

If both *k* and *threshold* are set, only features satisfying both conditions will be selected.

If *method* is not set, it is automatically selected when presented with the data set. Data sets with both continuous and discrete features are scored using a method suitable for the majority of features.

Parameters **method** : Orange.preprocess.score.ClassificationScorer, Orange.preprocess.score.SklScorer

Univariate feature scoring method.

k : int

The number of top features to select.

threshold : float

A threshold that a feature should meet according to the provided method.

decreasing : boolean

The order of feature importance when sorted from the most to the least important feature.

Preprocessors

Classification (`classification`)

Logistic Regression

class `Orange.classification.`**`LogisticRegressionLearner`**(*penalty='l2'*, *dual=False*, *tol=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1*, *class_weight=None*, *random_state=None*, *solver='liblinear'*, *max_iter=100*, *multi_class='ovr'*, *verbose=0*, *n_jobs=1*, *preprocessors=None*)

A wrapper for *sklearn.linear_model.logistic.LogisticRegression*. The following is its documentation:

Logistic Regression (aka logit, MaxEnt) classifier.

In the multiclass case, the training algorithm uses the one-vs-rest (OvR) scheme if the 'multi_class' option is set to 'ovr', and uses the cross- entropy loss if the 'multi_class' option is set to 'multinomial'. (Currently the 'multinomial' option is supported only by the 'lbfgs', 'sag' and 'newton-cg' solvers.)

This class implements regularized logistic regression using the 'liblinear' library, 'newton-cg', 'sag' and 'lbfgs' solvers. It can handle both dense and sparse input. Use C-ordered arrays or CSR matrices containing 64-bit floats for optimal performance; any other input format will be converted (and copied).

The 'newton-cg', 'sag', and 'lbfgs' solvers support only L2 regularization with primal formulation. The 'liblinear' solver supports both L1 and L2 regularization, with a dual formulation only for the L2 penalty.

Read more in the User Guide.

Random Forest

class `Orange.classification.`**`RandomForestLearner`**(*n_estimators=10*, *criterion='gini'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *max_leaf_nodes=None*, *bootstrap=True*, *oob_score=False*, *n_jobs=1*, *random_state=None*, *verbose=0*, *class_weight=None*, *preprocessors=None*)

A wrapper for *sklearn.ensemble.forest.RandomForestClassifier*. The following is its documentation:

A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement if *bootstrap=True* (default).

Read more in the User Guide.

Simple Random Forest

class Orange.classification.**SimpleRandomForestLearner**(*n_estimators=10*, *min_instances=2*, *max_depth=1024*, *max_majority=1.0*, *skip_prob='sqrt'*, *seed=42*)

A random forest classifier, optimized for speed. Trees in the forest are constructed with *SimpleTreeLearner* classification trees.

Parameters **n_estimators** : int, optional (default = 10)

Number of trees in the forest.

min_instances : int, optional (default = 2)

Minimal number of data instances in leaves. When growing the three, new nodes are not introduced if they would result in leaves with fewer instances than min_instances. Instance count is weighed.

max_depth : int, optional (default = 1024)

Maximal depth of tree.

max_majority : float, optional (default = 1.0)

Maximal proportion of majority class. When this is exceeded, induction stops (only used for classification).

skip_prob : string, optional (default = "sqrt")

Data attribute will be skipped with probability `skip_prob`.

- if float, then skip attribute with this probability.
- if "sqrt", then *skip_prob = 1 - sqrt(n_features) / n_features*
- if "log2", then *skip_prob = 1 - log2(n_features) / n_features*

seed : int, optional (default = 42)

Random seed.

Softmax Regression

class Orange.classification.**SoftmaxRegressionLearner**(*lambda_=1.0*, *preprocessors=None*, ***fmin_args*)

L2 regularized softmax regression classifier. Uses the L-BFGS algorithm to minimize the categorical cross entropy cost with L2 regularization. This model is suitable when dealing with a multi-class classification problem.

When using this learner you should:

•choose a suitable regularization parameter lambda_,

•consider using many logistic regression models (one for each value of the class variable) instead of softmax regression.

Parameters **lambda_** : float, optional (default=1.0)

Regularization parameter. It controls trade-off between fitting the data and keeping parameters small. Higher values of lambda_ force parameters to be smaller.

preprocessors : list, optional (default=[RemoveNaNClasses(), RemoveNaNColumns(), Impute(), Continuize(), Normalize()])

Preprocessors are applied to data before training or testing. Default preprocessors:

- remove columns with all values as NaN
- replace NaN values with suitable values
- continuize all discrete attributes,
- transform the dataset so that the columns are on a similar scale,

fmin_args : dict, optional

Parameters for L-BFGS algorithm.

k-Nearest Neighbors

class Orange.classification.**KNNLearner**(*n_neighbors=5*, *metric='euclidean'*, *weights='uniform'*, *algorithm='auto'*, *metric_params=None*, *preprocessors=None*)

A wrapper for *sklearn.neighbors.classification.KNeighborsClassifier*. The following is its documentation:

Classifier implementing the k-nearest neighbors vote.

Read more in the User Guide.

Naive Bayes

class Orange.classification.**NaiveBayesLearner**(*preprocessors=None*)

Naive Bayes classifier. Works only with discrete attributes. By default, continuous attributes are discretized.

Parameters **preprocessors** : list, optional (default="[Orange.preprocess.Discretize]")

An ordered list of preprocessors applied to data before training or testing.

The following code loads lenses data set (four discrete attributes and discrete class), constructs naive Bayesian learner, uses it on the entire data set to construct a classifier, and then applies classifier to the first three data instances:

```
>>> import Orange
>>> lenses = Orange.data.Table('lenses')
>>> nb = Orange.classification.NaiveBayesLearner()
>>> classifier = nb(lenses)
>>> classifier(lenses[0:3], True)
array([[ 0.04358755,  0.82671726,  0.12969519],
       [ 0.17428279,  0.20342097,  0.62229625],
       [ 0.18633359,  0.79518516,  0.01848125]])
```

For data sets that include continuous attributes,

Support Vector Machines

class Orange.classification.**SVMLearner**(*C=1.0*, *kernel='rbf'*, *degree=3*, *gamma='auto'*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *max_iter=-1*, *preprocessors=None*)

A wrapper for *sklearn.svm.classes.SVC*. The following is its documentation:

C-Support Vector Classification.

The implementation is based on libsvm. The fit time complexity is more than quadratic with the number of samples which makes it hard to scale to dataset with more than a couple of 10000 samples.

The multiclass support is handled according to a one-vs-one scheme.

For details on the precise mathematical formulation of the provided kernel functions and how *gamma*, *coef0* and *degree* affect each other, see the corresponding section in the narrative documentation: svm_kernels.

Read more in the User Guide.

Linear Support Vector Machines

class Orange.classification.**LinearSVMLearner**(*penalty='l2'*, *loss='squared_hinge'*, *dual=True*, *tol=0.0001*, *C=1.0*, *multi_class='ovr'*, *fit_intercept=True*, *intercept_scaling=True*, *random_state=None*, *preprocessors=None*)

A wrapper for *sklearn.svm.classes.LinearSVC*. The following is its documentation:

Linear Support Vector Classification.

Similar to SVC with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

This class supports both dense and sparse input and the multiclass support is handled according to a one-vs-the-rest scheme.

Read more in the User Guide.

Nu-Support Vector Machines

class Orange.classification.**NuSVMLearner**(*nu=0.5*, *kernel='rbf'*, *degree=3*, *gamma='auto'*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *max_iter=-1*, *preprocessors=None*)

A wrapper for *sklearn.svm.classes.NuSVC*. The following is its documentation:

Nu-Support Vector Classification.

Similar to SVC but uses a parameter to control the number of support vectors.

The implementation is based on libsvm.

Read more in the User Guide.

One Class Support Vector Machines

class Orange.classification.**OneClassSVMLearner**(*kernel='rbf'*, *degree=3*, *gamma='auto'*, *coef0=0.0*, *tol=0.001*, *nu=0.5*, *shrinking=True*, *cache_size=200*, *max_iter=-1*, *preprocessors=None*)

A wrapper for *sklearn.svm.classes.OneClassSVM*. The following is its documentation:

Unsupervised Outlier Detection.

Estimate the support of a high-dimensional distribution.

The implementation is based on libsvm.

Read more in the User Guide.

Classification Tree

Orange includes three implemenations of classification trees. *TreeLearner* is home-grown and properly handles multinominal and missing values. The one from scikit-learn, *SklTreeLearner*, is faster. Another home-grown, *SimpleTreeLearner*, is simpler and stil faster.

class Orange.classification.**TreeLearner**(**args*, *binarize=False*, *max_depth=None*, *min_samples_leaf=1*, *min_samples_split=2*, *sufficient_majority=0.95*, *preprocessors=None*, ***kwargs*)

Tree inducer with proper handling of nominal attributes and binarization.

The inducer can handle missing values of attributes and target. For discrete attributes with more than two possible values, each value can get a separate branch (*binarize=False*), or values can be grouped into two groups (*binarize=True*, default).

The tree growth can be limited by the required number of instances for internal nodes and for leafs, the sufficient proportion of majority class, and by the maximal depth of the tree.

If the tree is not binary, it can contain zero-branches.

Args:

binarize (bool): if *True* the inducer will find optimal split into two subsets for values of discrete attributes. If *False* (default), each value gets its branch.

min_samples_leaf (float): the minimal number of data instances in a leaf

min_samples_split (float): the minimal nubmer of data instances that is split into subgroups

max_depth (int): the maximal depth of the tree

sufficient_majority (float): a majority at which the data is not split further

Returns: instance of OrangeTreeModel

build_tree(*data*, *active_inst*, *level=1*)

Induce a tree from the given data

Returns: root node (Node)

class Orange.classification.**SklTreeLearner**(*criterion='gini'*, *splitter='best'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *max_features=None*, *random_state=None*, *max_leaf_nodes=None*, *preprocessors=None*)

Wrapper for SKL's tree inducer

Simple Tree

class Orange.classification.**SimpleTreeLearner**(*min_instances=2*, *max_depth=1024*, *max_majority=1.0*, *skip_prob=0.0*, *bootstrap=False*, *seed=42*)

Classification or regression tree learner. Uses gain ratio for classification and mean square error for regression. This learner was developed to speed-up random forest construction, but can also be used as a standalone tree learner.

min_instances [int, optional (default = 2)] Minimal number of data instances in leaves. When growing the three, new nodes are not introduced if they would result in leaves with fewer instances than min_instances. Instance count is weighed.

max_depth [int, optional (default = 1024)] Maximal depth of tree.

max_majority [float, optional (default = 1.0)] Maximal proportion of majority class. When this is exceeded, induction stops (only used for classification).

skip_prob [string, optional (default = 0.0)] Data attribute will be skipped with probability `skip_prob`.

- if float, then skip attribute with this probability.
- if "sqrt", then *skip_prob = 1 - sqrt(n_features) / n_features*
- if "log2", then *skip_prob = 1 - log2(n_features) / n_features*

bootstrap [data table, optional (default = False)] A bootstrap data set.

seed [int, optional (default = 42)] Random seed.

Majority Classifier

class Orange.classification.**MajorityLearner**(*preprocessors=None*)

A majority classifier. Always returns most frequent class from the training set, regardless of the attribute values from the test data instance. Returns class value distribution if class probabilities are requested. Can be used as a baseline when comparing classifiers.

In the special case of uniform class distribution within the training data, class value is selected randomly. In order to produce consistent results on the same data set, this value is selected based on hash of the class vector.

Elliptic Envelope

class Orange.classification.**EllipticEnvelopeLearner**(*store_precision=True*, *assume_centered=False*, *support_fraction=None*, *contamination=0.1*, *random_state=None*, *preprocessors=None*)

A wrapper for *sklearn.covariance.outlier_detection.EllipticEnvelope*. The following is its documentation:

An object for detecting outliers in a Gaussian distributed dataset.

Read more in the User Guide.

Neural Network

class Orange.classification.**NNClassificationLearner**(*hidden_layer_sizes=(100,)*, *activation='relu'*, *solver='adam'*, *alpha=0.0001*, *batch_size='auto'*, *learning_rate='constant'*, *learning_rate_init=0.001*, *power_t=0.5*, *max_iter=200*, *shuffle=True*, *random_state=None*, *tol=0.0001*, *verbose=False*, *warm_start=False*, *momentum=0.9*, *nesterovs_momentum=True*, *early_stopping=False*, *validation_fraction=0.1*, *beta_1=0.9*, *beta_2=0.999*, *epsilon=1e-08*, *preprocessors=None*)

A wrapper for *sklearn.neural_network.multilayer_perceptron.MLPClassifier*. The following is its documentation:

Multi-layer Perceptron classifier.

This model optimizes the log-loss function using LBFGS or stochastic gradient descent.

New in version 0.18.

CN2 Rule Induction

Induction of rules works by finding a rule that covers some learning instances, removing these instances, and repeating this until all instances are covered. Rules are scored by heuristics such as impurity of class distribution of covered instances. The module includes common rule-learning algorithms, and allows for replacing rule search strategies, scoring and other components.

class Orange.classification.rules.**CN2Learner**(*preprocessors=None*, *base_rules=None*)

Classic CN2 inducer that constructs a list of ordered rules. To evaluate found hypotheses, entropy measure is used. Returns a CN2Classifier if called with data.

References

[R11]

class Orange.classification.rules.**CN2UnorderedLearner**(*preprocessors=None*, *base_rules=None*)

Construct a set of unordered rules.

Rules are learnt for each class individually and scored by the relative frequency of the class corrected by the Laplace correction. After adding a rule, only the covered examples of that class are removed.

The code below loads the *iris* data set (four continuous attributes and a discrete class) and fits the learner.

```
import Orange
data = Orange.data.Table('iris')
learner = Orange.classification.CN2UnorderedLearner()

# consider up to 10 solution streams at one time
learner.rule_finder.search_algorithm.beam_width = 10

# continuous value space is constrained to reduce computation time
```

```
learner.rule_finder.search_strategy.constrain_continuous = True

# found rules must cover at least 15 examples
learner.rule_finder.general_validator.min_covered_examples = 15

# found rules may combine at most 2 selectors (conditions)
learner.rule_finder.general_validator.max_rule_length = 2

classifier = learner(data)
```

References

[R33]

class Orange.classification.rules.**CN2SDLearner**(*preprocessors=None*, *base_rules=None*)
Ordered CN2SD inducer that constructs a list of ordered rules. To evaluate found hypotheses, Weighted relative accuracy measure is used. Returns a CN2SDClassifier if called with data.

In this setting, ordered rule induction refers exclusively to finding best rule conditions and assigning the majority class in the rule head (target class is set to None). To later predict instances, rules will be regarded as unordered.

Notes

A weighted covering algorithm is applied, in which subsequently induced rules also represent interesting and sufficiently large subgroups of the population. Covered positive examples are not deleted from the learning set, rather their weight is reduced.

The algorithm demonstrates how classification rule learning (predictive induction) can be adapted to subgroup discovery, a task at the intersection of predictive and descriptive induction.

References

[R55]

class Orange.classification.rules.**CN2SDUnorderedLearner**(*preprocessors=None*, *base_rules=None*)
Unordered CN2SD inducer that constructs a set of unordered rules. To evaluate found hypotheses, Weighted relative accuracy measure is used. Returns a CN2SDUnorderedClassifier if called with data.

Notes

A weighted covering algorithm is applied, in which subsequently induced rules also represent interesting and sufficiently large subgroups of the population. Covered positive examples are not deleted from the learning set, rather their weight is reduced.

The algorithm demonstrates how classification rule learning (predictive induction) can be adapted to subgroup discovery, a task at the intersection of predictive and descriptive induction.

References

[R77]

Regression (regression)

Linear Regression

Linear regression is a statistical regression method which tries to predict a value of a continuous response (class) variable based on the values of several predictors. The model assumes that the response variable is a linear combination of the predictors, the task of linear regression is therefore to fit the unknown coefficients.

Example

```
>>> from Orange.regression.linear import LinearRegressionLearner
>>> mpg = Orange.data.Table('auto-mpg')
>>> mean_ = LinearRegressionLearner()
>>> model = mean_(mpg[40:110])
>>> print(model)
LinearModel LinearRegression(copy_X=True, fit_intercept=True, normalize=False)
>>> mpg[20]
Value('mpg', 25.0)
>>> model(mpg[0])
Value('mpg', 24.6)
```

class Orange.regression.linear.**LinearRegressionLearner**(*preprocessors=None*)

A wrapper for *sklearn.linear_model.base.LinearRegression*. The following is its documentation:

Ordinary least squares Linear Regression.

class Orange.regression.linear.**RidgeRegressionLearner**(*alpha=1.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=None*, *tol=0.001*, *solver='auto'*, *preprocessors=None*)

A wrapper for *sklearn.linear_model.ridge.Ridge*. The following is its documentation:

Linear least squares with l2 regularization.

This model solves a regression model where the loss function is the linear least squares function and regularization is given by the l2-norm. Also known as Ridge Regression or Tikhonov regularization. This estimator has built-in support for multi-variate regression (i.e., when y is a 2d-array of shape [n_samples, n_targets]).

Read more in the User Guide.

class Orange.regression.linear.**LassoRegressionLearner**(*alpha=1.0*, *fit_intercept=True*, *normalize=False*, *precompute=False*, *copy_X=True*, *max_iter=1000*, *tol=0.0001*, *warm_start=False*, *positive=False*, *preprocessors=None*)

A wrapper for *sklearn.linear_model.coordinate_descent.Lasso*. The following is its documentation:

Linear Model trained with L1 prior as regularizer (aka the Lasso)

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

Technically the Lasso model is optimizing the same objective function as the Elastic Net with `l1_ratio=1.0` (no L2 penalty).

Read more in the User Guide.

class Orange.regression.linear.**SGDRegressionLearner**(*loss='squared_loss'*, *penalty='l2'*, *alpha=0.0001*, *l1_ratio=0.15*, *fit_intercept=True*, *n_iter=5*, *shuffle=True*, *epsilon=0.1*, *n_jobs=1*, *random_state=None*, *learning_rate='invscaling'*, *eta0=0.01*, *power_t=0.25*, *class_weight=None*, *warm_start=False*, *average=False*, *preprocessors=None*)

A wrapper for *sklearn.linear_model.stochastic_gradient.SGDRegressor*. The following is its documentation:

Linear model fitted by minimizing a regularized empirical loss with SGD

SGD stands for Stochastic Gradient Descent: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate).

The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieve online feature selection.

This implementation works with data represented as dense numpy arrays of floating point values for the features.

Read more in the User Guide.

class Orange.regression.linear.**LinearModel**(*skl_model*)

Polynomial

Polynomial model is a wrapper that constructs polynomial features of a specified degree and learns a model on them.

class Orange.regression.linear.**PolynomialLearner**(*learner=LinearRegressionLearner()*, *degree=2*, *preprocessors=None*)

Generate polynomial features and learn a prediction model

Parameters **learner** : LearnerRegression

learner to be fitted on the transformed features

degree : int

degree of used polynomial

preprocessors : List[Preprocessor]

preprocessors to be applied on the data before learning

Mean

Mean model predicts the same value (usually the distribution mean) for all data instances. Its accuracy can serve as a baseline for other regression models.

The model learner (*MeanLearner*) computes the mean of the given data or distribution. The model is stored as an instance of MeanModel.

Example

```
>>> from Orange.data import Table
>>> from Orange.regression import MeanLearner
>>> data = Table('auto-mpg')
>>> learner = MeanLearner()
>>> model = learner(data)
>>> print(model)
MeanModel(23.51457286432161)
>>> model(data[:4])
array([ 23.51457286,  23.51457286,  23.51457286,  23.51457286])
```

class Orange.regression.**MeanLearner**(*preprocessors=None*)

Fit a regression model that returns the average response (class) value.

fit_storage(*data*)

Construct a MeanModel by computing the mean value of the given data.

Parameters **data** (Orange.data.Table) – data table

Returns regression model, which always returns mean value

Return type MeanModel

Random Forest

class Orange.regression.**RandomForestRegressionLearner**(*n_estimators=10*, *criterion='mse'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *max_leaf_nodes=None*, *bootstrap=True*, *oob_score=False*, *n_jobs=1*, *random_state=None*, *verbose=0*, *preprocessors=None*)

A wrapper for *sklearn.ensemble.forest.RandomForestRegressor*. The following is its documentation:

A random forest regressor.

A random forest is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement if *bootstrap=True* (default).

Read more in the User Guide.

Simple Random Forest

class Orange.regression.**SimpleRandomForestLearner**(*n_estimators=10*, *min_instances=2*, *max_depth=1024*, *max_majority=1.0*, *skip_prob='sqrt'*, *seed=42*)

A random forest regressor, optimized for speed. Trees in the forest are constructed with SimpleTreeLearner classification trees.

Parameters **n_estimators** : int, optional (default = 10)

Number of trees in the forest.

min_instances : int, optional (default = 2)

Minimal number of data instances in leaves. When growing the three, new nodes are not introduced if they would result in leaves with fewer instances than min_instances. Instance count is weighed.

max_depth : int, optional (default = 1024)

Maximal depth of tree.

max_majority : float, optional (default = 1.0)

Maximal proportion of majority class. When this is exceeded, induction stops (only used for classification).

skip_prob : string, optional (default = "sqrt")

Data attribute will be skipped with probability `skip_prob`.

- if float, then skip attribute with this probability.
- if "sqrt", then *skip_prob = 1 - sqrt(n_features) / n_features*
- if "log2", then *skip_prob = 1 - log2(n_features) / n_features*

seed : int, optional (default = 42)

Random seed.

Regression Tree

Orange includes two implemenations of regression tres: a home-grown one, and one from scikit-learn. The former properly handles multinominal and missing values, and the latter is faster.

class `Orange.regression.`**`TreeLearner`**`(*args, binarize=False, min_samples_leaf=1, min_samples_split=2, max_depth=None, **kwargs)`

Tree inducer with proper handling of nominal attributes and binarization.

The inducer can handle missing values of attributes and target. For discrete attributes with more than two possible values, each value can get a separate branch (*binarize=False*), or values can be grouped into two groups (*binarize=True*, default).

The tree growth can be limited by the required number of instances for internal nodes and for leafs, and by the maximal depth of the tree.

If the tree is not binary, it can contain zero-branches.

Parameters **binarize**

if *True* the inducer will find optimal split into two subsets for values of discrete attributes. If *False* (default), each value gets its branch.

min_samples_leaf

the minimal number of data instances in a leaf

min_samples_split

the minimal number of data instances that is split into subgroups

max_depth

the maximal depth of the tree

Returns instance of OrangeTreeModel

build_tree(*data*, *active_inst*, *level=1*)
Induce a tree from the given data

Returns: root node (Node)

class Orange.regression.**SklTreeRegressionLearner**(*criterion='mse'*, *splitter='best'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *max_features=None*, *random_state=None*, *max_leaf_nodes=None*, *preprocessors=None*)

A wrapper for *sklearn.tree.tree.DecisionTreeRegressor*. The following is its documentation:

A decision tree regressor.

Read more in the User Guide.

Neural Network

class Orange.regression.**NNRegressionLearner**(*hidden_layer_sizes=(100,)*, *activation='relu'*, *solver='adam'*, *alpha=0.0001*, *batch_size='auto'*, *learning_rate='constant'*, *learning_rate_init=0.001*, *power_t=0.5*, *max_iter=200*, *shuffle=True*, *random_state=None*, *tol=0.0001*, *verbose=False*, *warm_start=False*, *momentum=0.9*, *nesterovs_momentum=True*, *early_stopping=False*, *validation_fraction=0.1*, *beta_1=0.9*, *beta_2=0.999*, *epsilon=1e-08*, *preprocessors=None*)

A wrapper for *sklearn.neural_network.multilayer_perceptron.MLPRegressor*. The following is its documentation:

Multi-layer Perceptron regressor.

This model optimizes the squared-loss using LBFGS or stochastic gradient descent.

New in version 0.18.

Clustering (`clustering`)

Hierarchical (`hierarchical`)

Example

The following example shows clustering of the Iris data with distance matrix computed with the `Orange.distance.Euclidean` distance and clustering using average linkage.

```
>>> from Orange import data, distance
>>> from Orange.clustering import hierarchical
>>> data = data.Table('iris')
>>> dist_matrix = distance.Euclidean(data)
```

```
>>> hierar = hierarchical.HierarchicalClustering(n_clusters=3)
>>> hierar.linkage = hierarchical.AVERAGE
>>> hierar.fit(dist_matrix)
>>> hierar.labels
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  2.,  0.,  2.,  2.,
        2.,  2.,  0.,  2.,  2.,  2.,  2.,  2.,  2.,  0.,  0.,  2.,  2.,
        2.,  2.,  0.,  2.,  0.,  2.,  0.,  2.,  2.,  0.,  0.,  2.,  2.,
        2.,  2.,  2.,  0.,  2.,  2.,  2.,  2.,  0.,  2.,  2.,  2.,  0.,
        2.,  2.,  2.,  0.,  2.,  2.,  0.])
```

Hierarchical Clustering

class `Orange.clustering.hierarchical.`**`HierarchicalClustering`**(*n_clusters=2*, *linkage='average'*)

Distance (`distance`)

The following example demonstrates how to compute distances between all data instances from Iris:

```
>>> from Orange.data import Table
>>> from Orange.distance import Euclidean
>>> iris = Table('iris')
>>> dist_matrix = Euclidean(iris)
>>> # Distance between first two examples
>>> dist_matrix.X[0, 1]
0.53851648
```

To compute distances between all columns, we set *axis* to 0.

```
>>> Euclidean(iris, axis=0)
DistMatrix([[  0.        ,  36.17927584,  28.9542743 ,  57.1913455 ],
            [ 36.17927584,   0.        ,  25.73382987,  25.81259383],
            [ 28.9542743 ,  25.73382987,   0.        ,  33.87270287],
            [ 57.1913455 ,  25.81259383,  33.87270287,   0.        ]])
```

Finally, we can compute distances between all pairs of rows from two tables.

```
>>> iris1 = iris[:100]
>>> iris2 = iris[100:]
>>> dist = Euclidean(iris_even, iris_odd)
>>> dist.shape
(75, 100)
```

Most metrics can be fit on training data to normalize values and handle missing data. We do so by calling the constructor without arguments or with parameters, such as *normalize*, and then pass the data to method *fit*.

```
>>> dist_model = Euclidean(normalize=True).fit(iris1)
>>> dist = dist_model(iris2[:3])
>>> dist
DistMatrix([[ 0.        ,  1.36778277,  1.11352233],
            [ 1.36778277,  0.        ,  1.57810546],
            [ 1.11352233,  1.57810546,  0.        ]])
```

The above distances are computed on the first three rows of *iris2*, normalized by means and variances computed from *iris1*.

Here are five closest neighbors of *iris2[0]* from *iris1*:

```
>>> dist0 = dist_model(iris1, iris2[0])
>>> neigh_idx = np.argsort(dist0.flatten())[:5]
>>> iris1[neigh_idx]
[[5.900, 3.200, 4.800, 1.800 | Iris-versicolor],
 [6.700, 3.000, 5.000, 1.700 | Iris-versicolor],
 [6.300, 3.300, 4.700, 1.600 | Iris-versicolor],
 [6.000, 3.400, 4.500, 1.600 | Iris-versicolor],
 [6.400, 3.200, 4.500, 1.500 | Iris-versicolor]
]
```

All distances share a common interface.

class `Orange.distance.`**`Distance`**

Base class for construction of distances models (`DistanceModel`).

Distances can be computed between all pairs of rows in one table, or between pairs where one row is from one table and one from another.

If *axis* is set to *0*, the class computes distances between all pairs of columns in a table. Distances between columns from separate tables are probably meaningless, thus unsupported.

The class can be used as follows:

- Constructor is called only with keyword argument *axis* that specifies the axis over which the distances are computed, and with other subclass-specific keyword arguments.
- Next, we call the method *fit(data)* to produce an instance of `DistanceModel`; the instance stores any parameters needed for computation of distances, such as statistics for normalization and handling of missing data.
- We can then call the `DistanceModel` with data to compute the distance between its rows or columns, or with two data tables to compute distances between all pairs of rows.

The second, shorter way to use this class is to call the constructor with one or two data tables and any additional keyword arguments. Constructor will execute the above steps and return `DistMatrix`. Such usage is here for backward compatibility, practicality and efficiency.

Args:

e1 (*`Table`* or *`Instance`* or `np.ndarray` or *None*): data on which to train the model and compute the distances

e2 (*`Table`* or *`Instance`* or `np.ndarray` or *None*): if present, the class computes distances with pairs coming from the two tables

axis (int): axis over which the distances are computed, 1 (default) for rows, 0 for columns

impute (bool): if *True* (default is *False*), nans in the computed distances are replaced with zeros, and infs with very large numbers.

Attributes:

axis (int): axis over which the distances are computed, 1 (default) for rows, 0 for columns

impute (bool): if *True* (default is *False*), nans in the computed distances are replaced with zeros, and infs with very large numbers.

The capabilities of the metrics are described with class attributes.

If class attribute *supports_discrete* is *True*, the distance also uses discrete attributes to compute row distances. The use of discrete attributes depends upon the type of distance; e.g. Jaccard distance observes whether the value is zero or non-zero, while Euclidean and Manhattan distance observes whether a pair of values is same or different.

Class attribute *supports_missing* indicates that the distance can cope with missing data. In such cases, letting the distance handle it should be preferred over pre-imputation of missing values.

Class attribute *supports_normalization* indicates that the constructor accepts an argument *normalize*. If set to *True*, the metric will attempt to normalize the values in a sense that each attribute will have equal influence. For instance, the Euclidean distance subtract the mean and divides the result by the deviation, while Manhattan distance uses the median and MAD.

If class attribute *supports_sparse* is *True*, the class will handle sparse data. Currently, all classes that do handle it rely on fallbacks to SKL metrics. These, however, do not support discrete data and missing values, and will fail silently.

Handling discrete and missing data

Discrete data is handled as appropriate for the particular distance. For instance, the Euclidean distance treats a pair of values as either the same or different, contributing either 0 or 1 to the squared sum of differences. In other cases – particularly in Jaccard and cosine distance, discrete values are treated as zero or non-zero.

Missing data is not simply imputed. We assume that values of each variable are distributed by some unknown distribution and compute - without assuming a particular distribution shape - the expected distance. For instance, for the Euclidean distance it turns out that the expected squared distance between a known and a missing value equals the square of the known value's distance from the mean of the missing variable, plus its variance.

Supported distances

Euclidean distance

For numeric values, the Euclidean distance is the square root of sums of squares of pairs of values from rows or columns. For discrete values, 1 is added if the two values are different.

To put all numeric data on the same scale, and in particular when working with a mixture of numeric and discrete data, it is recommended to enable normalization by adding *normalize=True* to the constructor. With this, numeric values are normalized by subtracting their mean and divided by deviation multiplied by the square root of two. The mean and deviation are computed on the training data, if the *fit* method is used. When computing distances between two tables and without explicitly calling *fit*, means and variances are computed from the first table only. Means and variances are always computed from columns, disregarding the axis over which we compute the distances, since columns represent variables and hence come from a certain distribution.

As described above, the expected squared difference between a known and a missing value equals the squared difference between the known value and the mean, plus the variance. The squared difference between two unknown values equals twice the variance.

For normalized data, the difference between a known and missing numeric value equals the square of the known value + 0.5. The difference between two missing values is 1.

For discrete data, the expected difference between a known and a missing value equals the probablity that the two values are different, which is 1 minus the probability of the known value. If both values are missing, the probability of them being different equals 1 minus the sum of squares of all probabilities (also known as the Gini index).

Manhattan distance

Manhattan distance is the sum of absolute pairwise distances.

Normalization and treatment of missing values is similar as in the Euclidean distance, except that medians and median absolute distance from the median (MAD) are used instead of means and deviations.

For discrete values, distances are again 0 or 1, hence the Manhattan distance for discrete columns is the same as the Euclidean.

Cosine distance

Cosine similarity is the dot product divided by the product of lengths (where the length is the square of dot product of a row/column with itself). Cosine distance is computed by subtracting the similarity from one.

In calculation of dot products, missing values are replaced by means. In calculation of lengths, the contribution of a missing value equals the square of the mean plus the variance. (The difference comes from the fact that in the former case the missing values are independent.)

Non-zero discrete values are replaced by 1. This introduces the notion of a "base value", which is the first in the list of possible values. In most cases, this will only make sense for indicator (i.e. two-valued, boolean attributes).

Cosine distance does not support any column-wise normalization.

Jaccard distance

Jaccard similarity between two sets is defined as the size of their intersection divided by the size of the union. Jaccard distance is computed by subtracting the similarity from one.

In Orange, attribute values are interpreted as membership indicator. In row-wise distances, columns are interpreted as sets, and non-zero values in a row (including negative values of numeric features) indicate that the row belongs to the particular sets. In column-wise distances, rows are sets and values indicate the sets to which the column belongs.

For missing values, relative frequencies from the training data are used as probabilities for belonging to a set. That is, for row-wise distances, we compute the relative frequency of non-zero values in each column, and vice-versa for column-wise distances. For intersection (union) of sets, we then add the probability of belonging to both (any of) the two sets instead of adding a 0 or 1.

SpearmanR, AbsoluteSpearmanR, PearsonR, AbsolutePearsonR

The four correlation-based distance measure equal (1 - the correlation coefficient) / 2. For *AbsoluteSpearmanR* and *AbsolutePearsonR*, the absolute value of the coefficient is used.

These distances do not handle missing or discrete values.

Mahalanobis distance

Mahalanobis distance is similar to cosine distance, except that the data is projected into the PCA space.

Mahalanobis distance does not handle missing or discrete values.

Evaluation (evaluation)

Sampling procedures for testing models (testing)

class Orange.evaluation.testing.**Results**(*data=None*, *nmethods=0*, ***, *learners=None*, *train_data=None*, *nrows=None*, *nclasses=None*, *store_data=False*, *store_models=False*, *domain=None*, *actual=None*, *row_indices=None*, *predicted=None*, *probabilities=None*, *preprocessor=None*, *callback=None*, *n_jobs=1*)

Class for storing predictions in model testing.

Attributes:

data (Optional[Table]): Data used for testing. When data is stored, this is typically not a copy but a reference.

models (Optional[List[Model]]): A list of induced models.

row_indices (np.ndarray): Indices of rows in *data* that were used in testing, stored as a numpy vector of length *nrows*. Values of *actual[i]*, *predicted[i]* and *probabilities[i]* refer to the target value of instance *data[row_indices[i]]*.

nrows (int): The number of test instances (including duplicates).

actual (np.ndarray): Actual values of target variable; a numpy vector of length *nrows* and of the same type as *data* (or *np.float32* if the type of data cannot be determined).

predicted (np.ndarray): Predicted values of target variable; a numpy array of shape (number-of-methods, *nrows*) and of the same type as *data* (or *np.float32* if the type of data cannot be determined).

probabilities (Optional[np.ndarray]): Predicted probabilities (for discrete target variables); a numpy array of shape (number-of-methods, *nrows*, number-of-classes) of type *np.float32*.

folds (List[Slice or List[int]]): A list of indices (or slice objects) corresponding to rows of each fold.

get_augmented_data(*model_names*, *include_attrs=True*, *include_predictions=True*, *include_probabilities=True*)

Return the data, augmented with predictions, probabilities (if the task is classification) and folds info. Predictions, probabilities and folds are inserted as meta attributes.

Args: model_names (list): A list of strings containing learners' names. include_attrs (bool): Flag that tells whether to include original attributes. include_predictions (bool): Flag that tells whether to include predictions. include_probabilities (bool): Flag that tells whether to include probabilities.

Returns: Orange.data.Table: Data augmented with predictions, (probabilities) and (fold).

fit(*train_data*, *test_data=None*)

Fits *self.learners* using folds sampled from the provided data.

Parameters **train_data** : Table

table to sample train folds

test_data : Optional[Table]

tap to sample test folds of None then *train_data* will be used

prepare_arrays(*test_data*)

Initialize arrays that will be used by *fit* method.

setup_indices(*train_data*, *test_data*)

Initializes *self.indices* with iterable objects with slices (or indices) for each fold.

Args: train_data (Table): train table test_data (Table): test table

split_by_model()
Split evaluation results by models

class Orange.evaluation.testing.**CrossValidation**(*data*, *learners*, *k=10*, *stratified=True*, *random_state=0*, *store_data=False*, *store_models=False*, *preprocessor=None*, *callback=None*, *warnings=None*, *n_jobs=1*)
K-fold cross validation.

If the constructor is given the data and a list of learning algorithms, it runs cross validation and returns an instance of *Results* containing the predicted values and probabilities.

k
The number of folds.

random_state

class Orange.evaluation.testing.**CrossValidationFeature**(*data*, *learners*, *feature*, *store_data=False*, *store_models=False*, *preprocessor=None*, *callback=None*, *n_jobs=1*)
Cross validation with folds according to values of a feature.

feature
The feature defining the folds.

class Orange.evaluation.testing.**LeaveOneOut**(*data*, *learners*, *store_data=False*, *store_models=False*, *preprocessor=None*, *callback=None*, *n_jobs=1*)
Leave-one-out testing

class Orange.evaluation.testing.**TestOnTestData**(*train_data*, *test_data*, *learners*, *store_data=False*, *store_models=False*, *preprocessor=None*, *callback=None*, *n_jobs=1*)
Test on a separate test data set.

class Orange.evaluation.testing.**TestOnTrainingData**(*data*, *learners*, *store_data=False*, *store_models=False*, *preprocessor=None*, *callback=None*, *n_jobs=1*)
Trains and test on the same data

Orange.evaluation.testing.**sample**(*table*, *n=0.7*, *stratified=False*, *replace=False*, *random_state=None*)
Samples data instances from a data table. Returns the sample and a data set from input data table that are not in the sample. Also uses several sampling functions from scikit-learn.

table [data table] A data table from which to sample.

n [float, int (default = 0.7)] If float, should be between 0.0 and 1.0 and represents the proportion of data instances in the resulting sample. If int, n is the number of data instances in the resulting sample.

stratified [bool, optional (default = False)] If true, sampling will try to consider class values and match distribution of class values in train and test subsets.

replace [bool, optional (default = False)] sample with replacement

random_state [int or RandomState] Pseudo-random number generator state used for random sampling.

Scoring methods (scoring)

CA

Orange.evaluation.**CA**(*cls*, *results=None*, ***kwargs*)

A wrapper for *sklearn.metrics.classification.accuracy_score*. The following is its documentation:

Accuracy classification score.

In multilabel classification, this function computes subset accuracy: the set of labels predicted for a sample must *exactly* match the corresponding set of labels in y_true.

Read more in the User Guide.

Precision

Orange.evaluation.**Precision**(*cls*, *results=None*, ***kwargs*)

A wrapper for *sklearn.metrics.classification.precision_score*. The following is its documentation:

Compute the precision

The precision is the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The best value is 1 and the worst value is 0.

Read more in the User Guide.

Recall

Orange.evaluation.**Recall**(*cls*, *results=None*, ***kwargs*)

A wrapper for *sklearn.metrics.classification.recall_score*. The following is its documentation:

Compute the recall

The recall is the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The best value is 1 and the worst value is 0.

Read more in the User Guide.

F1

Orange.evaluation.**F1**(*cls*, *results=None*, ***kwargs*)

A wrapper for *sklearn.metrics.classification.f1_score*. The following is its documentation:

Compute the F1 score, also known as balanced F-score or F-measure

The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the F1 score are equal. The formula for the F1 score is:

```
F1 = 2 * (precision * recall) / (precision + recall)
```

In the multi-class and multi-label case, this is the weighted average of the F1 score of each class.

Read more in the User Guide.

PrecisionRecallFSupport

Orange.evaluation.**PrecisionRecallFSupport**(*cls*, *results=None*, ***kwargs*)

A wrapper for *sklearn.metrics.classification.precision_recall_fscore_support*. The following is its documentation:

Compute precision, recall, F-measure and support for each class

The precision is the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The recall is the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The F-beta score can be interpreted as a weighted harmonic mean of the precision and recall, where an F-beta score reaches its best value at 1 and worst score at 0.

The F-beta score weights recall more than precision by a factor of `beta`. `beta == 1.0` means recall and precision are equally important.

The support is the number of occurrences of each class in `y_true`.

If `pos_label is None` and in binary classification, this function returns the average precision, recall and F-measure if `average` is one of `'micro'`, `'macro'`, `'weighted'` or `'samples'`.

Read more in the User Guide.

AUC

Orange.evaluation.**AUC**(*cls*, *results=None*, ***kwargs*)

${sklpar}

Parameters **results** : Orange.evaluation.Results

Stored predictions and actual data in model testing.

target : int, optional (default=None)

Value of class to report.

Log Loss

Orange.evaluation.**LogLoss**(*cls*, *results=None*, ***kwargs*)

${sklpar}

Parameters **results** : Orange.evaluation.Results

Stored predictions and actual data in model testing.

eps : float

Log loss is undefined for p=0 or p=1, so probabilities are clipped to max(eps, min(1 - eps, p)).

normalize : bool, optional (default=True)

If true, return the mean loss per sample. Otherwise, return the sum of the per-sample losses.

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

Examples

```
>>> Orange.evaluation.LogLoss(results)
array([ 0.3...])
```

MSE

Orange.evaluation.**MSE**(*cls*, *results=None*, ***kwargs*)

A wrapper for *sklearn.metrics.regression.mean_squared_error*. The following is its documentation:

Mean squared error regression loss

Read more in the User Guide.

MAE

Orange.evaluation.**MAE**(*cls*, *results=None*, ***kwargs*)

A wrapper for *sklearn.metrics.regression.mean_absolute_error*. The following is its documentation:

Mean absolute error regression loss

Read more in the User Guide.

R2

Orange.evaluation.**R2**(*cls*, *results=None*, ***kwargs*)

A wrapper for *sklearn.metrics.regression.r2_score*. The following is its documentation:

R^2 (coefficient of determination) regression score function.

Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Read more in the User Guide.

CD diagram

Orange.evaluation.**compute_CD**(*avranks*, *n*, *alpha='0.05'*, *test='nemenyi'*)

Returns critical difference for Nemenyi or Bonferroni-Dunn test according to given alpha (either alpha="0.05" or alpha="0.1") for average ranks and number of tested data sets N. Test can be either "nemenyi" for for Nemenyi two tailed test or "bonferroni-dunn" for Bonferroni-Dunn test.

Orange.evaluation.**graph_ranks**(*avranks*, *names*, *cd=None*, *cdmethod=None*, *lowv=None*, *highv=None*, *width=6*, *textspace=1*, *reverse=False*, *filename=None*, ***kwargs*)

Draws a CD graph, which is used to display the differences in methods' performance. See Janez Demsar, Statistical Comparisons of Classifiers over Multiple Data Sets, 7(Jan):1–30, 2006.

Needs matplotlib to work.

The image is ploted on *plt* imported using *import matplotlib.pyplot as plt*.

Args: avranks (list of float): average ranks of methods. names (list of str): names of methods. cd (float): Critical difference used for statistically significance of

difference between methods.

cdmethod (int, optional): the method that is compared with other methods If omitted, show pairwise comparison of methods

lowv (int, optional): the lowest shown rank highv (int, optional): the highest shown rank width (int, optional): default width in inches (default: 6) textspace (int, optional): space on figure sides (in inches) for the

method names (default: 1)

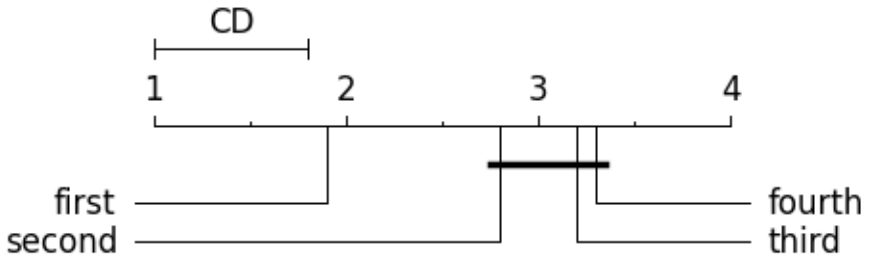
reverse (bool, optional): if set to *True*, the lowest rank is on the right (default: *False*)

filename (str, optional): output file name (with extension). If not given, the function does not write a file.

Example

```
>>> import Orange
>>> import matplotlib.pyplot as plt
>>> names = ["first", "third", "second", "fourth" ]
>>> avranks =  [1.9, 3.2, 2.8, 3.3 ]
>>> cd = Orange.evaluation.compute_CD(avranks, 30) #tested on 30 datasets
>>> Orange.evaluation.graph_ranks(avranks, names, cd=cd, width=6, textspace=1.5)
>>> plt.show()
```

The code produces the following graph:



Projection (`projection`)

PCA

Principal component analysis is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components.

Example

```
>>> from Orange.projection.pca import PCA
>>> from Orange.data import Table
>>> iris = Table('iris')
>>> pca = PCA()
>>> model = pca(iris)
>>> model.components_    # PCA components
array([[ 0.36158968, -0.08226889,  0.85657211,  0.35884393],
    [ 0.65653988,  0.72971237, -0.1757674 , -0.07470647],
    [-0.58099728,  0.59641809,  0.07252408,  0.54906091],
    [ 0.31725455, -0.32409435, -0.47971899,  0.75112056]])
>>> transformed_data = model(iris)    # transformed data
>>> transformed_data
[[-2.684, 0.327, -0.022, 0.001 | Iris-setosa],
[-2.715, -0.170, -0.204, 0.100 | Iris-setosa],
[-2.890, -0.137, 0.025, 0.019 | Iris-setosa],
[-2.746, -0.311, 0.038, -0.076 | Iris-setosa],
[-2.729, 0.334, 0.096, -0.063 | Iris-setosa],
...
]
```

class Orange.projection.pca.**PCA**(*n_components=None*, *copy=True*, *whiten=False*, *svd_solver='auto'*, *tol=0.0*, *iterated_power='auto'*, *random_state=None*, *preprocessors=None*)

A wrapper for *sklearn.decomposition.pca.PCA*. The following is its documentation:

Principal component analysis (PCA)

Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space.

It uses the LAPACK implementation of the full SVD or a randomized truncated SVD by the method of Halko et al. 2009, depending on the shape of the input data and the number of components to extract.

It can also use the scipy.sparse.linalg ARPACK implementation of the truncated SVD.

Notice that this class does not support sparse input. See TruncatedSVD for an alternative with sparse data.

Read more in the User Guide.

class Orange.projection.pca.**SparsePCA**(*n_components=None*, *alpha=1*, *ridge_alpha=0.01*, *max_iter=1000*, *tol=1e-08*, *method='lars'*, *n_jobs=1*, *U_init=None*, *V_init=None*, *verbose=False*, *random_state=None*, *preprocessors=None*)

A wrapper for *sklearn.decomposition.sparse_pca.SparsePCA*. The following is its documentation:

Sparse Principal Components Analysis (SparsePCA)

Finds the set of sparse components that can optimally reconstruct the data. The amount of sparseness is controllable by the coefficient of the L1 penalty, given by the parameter alpha.

Read more in the User Guide.

class Orange.projection.pca.**IncrementalPCA**(*n_components=None*, *whiten=False*, *copy=True*, *batch_size=None*, *preprocessors=None*)

A wrapper for *sklearn.decomposition.incremental_pca.IncrementalPCA*. The following is its documentation:

Incremental principal components analysis (IPCA).

Linear dimensionality reduction using Singular Value Decomposition of centered data, keeping only the most significant singular vectors to project the data to a lower dimensional space.

Depending on the size of the input data, this algorithm can be much more memory efficient than a PCA.

This algorithm has constant memory complexity, on the order of `batch_size`, enabling use of np.memmap files without loading the entire file into memory.

The computational overhead of each SVD is `O(batch_size * n_features ** 2)`, but only 2 * batch_size samples remain in memory at a time. There will be `n_samples / batch_size` SVD computations to get the principal components, versus 1 large SVD of complexity `O(n_samples * n_features ** 2)` for PCA.

Read more in the User Guide.

Miscellaneous (`misc`)

Distance Matrix (`distmatrix`)

class `Orange.misc.distmatrix.`**DistMatrix**
Distance matrix. Extends `numpy.ndarray`.

`row_items`
Items corresponding to matrix rows.

`col_items`
Items corresponding to matrix columns.

`axis`
If axis=1 we calculate distances between rows, if axis=0 we calculate distances between columns.

`dim`
Returns the single dimension of the symmetric square matrix.

`submatrix`(*row_items*, *col_items=None*)
Return a submatrix

Args: row_items: indices of rows col_items: incides of columns

classmethod `from_file`(*filename*)
Load distance matrix from a file

The file should be preferrably encoded in ascii/utf-8. White space at the beginning and end of lines is ignored.

The first line of the file starts with the matrix dimension. It can be followed by a list flags

- *axis=<number>*: the axis number
- *symmetric*: the matrix is symmetric; when reading the element (i, j) it's value is also assigned to (j, i)
- *asymmetric*: the matrix is asymmetric
- *row_labels*: the file contains row labels
- *col_labels*: the file contains column labels

By default, matrices are symmetric, have axis 1 and no labels are given. Flags *labeled* and *labelled* are obsolete aliases for *row_labels*.

If the file has column labels, they follow in the second line. Row labels appear at the beginning of each row. Labels are arbitrary strings that cannot contain newlines and tabulators. Labels are stored as instances of *Table* with a single meta attribute named "label".

The remaining lines contain tab-separated numbers, preceded with labels, if present. Lines are padded with zeros if necessary. If the matrix is symmetric, the file contains the lower triangle; any data above the diagonal is ignored.

Args: filename: file name

has_row_labels()
Returns *True* if row labels can be automatically determined from data

For this, the *row_items* must be an instance of *Orange.data.Table* whose domain contains a single meta attribute, which has to be a string. The domain may contain other variables, but not meta attributes.

has_col_labels()
Returns *True* if column labels can be automatically determined from data

For this, the *col_items* must be an instance of *Orange.data.Table* whose domain contains a single meta attribute, which has to be a string. The domain may contain other variables, but not meta attributes.

save(*filename*)
Save the distance matrix to a file in the file format described at *from_file*.

Args: filename: file name

Bibliography

[Quinlan198689] J R Quinlan: Induction of Decision Trees, Machine Learning, 1986.

[R11] "The CN2 Induction Algorithm", Peter Clark and Tim Niblett, Machine Learning Journal, 3 (4), pp261-283, (1989)

[R33] "Rule Induction with CN2: Some Recent Improvements", Peter Clark and Robin Boswell, Machine Learning - Proceedings of the 5th European Conference (EWSL-91), pp151-163, 1991

[R55] "Subgroup Discovery with CN2-SD", Nada Lavrač et al., Journal of Machine Learning Research 5 (2004), 153-188, 2004

[R77] "Subgroup Discovery with CN2-SD", Nada Lavrač et al., Journal of Machine Learning Research 5 (2004), 153-188, 2004

Python Module Index

O

Orange.classification, 49
Orange.classification.rules, 55
Orange.clustering, 61
Orange.data.filter, 35
Orange.data.variable, 32
Orange.evaluation, 66
Orange.evaluation.testing, 66
Orange.misc, 73
Orange.misc.distmatrix, 73
Orange.projection, 71
Orange.regression, 57

Index

Symbols

.. index:: linear fitter, 57
__bool__() (Orange.data.sql.table.SqlTable method), 23
__contains__() (Orange.data.Domain method), 26
__getitem__() (Orange.data.Domain method), 25
__getitem__() (Orange.data.sql.table.SqlTable method), 23
__getitem__() (in module Orange.data.storage), 16
__init__() (Orange.data.Domain method), 24
__init__() (Orange.data.sql.table.SqlTable method), 22
__iter__() (Orange.data.sql.table.SqlTable method), 23
__len__() (Orange.data.Domain method), 26
__len__() (Orange.data.sql.table.SqlTable method), 23
__len__() (Orange.data.storage. method), 16
_compute_contingency() (Orange.data.storage.Storage method), 18
_compute_distributions() (in module Orange.data.storage), 18
_filter_has_class() (in module Orange.data.storage), 17
_filter_is_defined() (in module Orange.data.storage), 17
_filter_same_value() (in module Orange.data.storage), 17
_filter_values() (in module Orange.data.storage), 17

A

adjust_decimals (Orange.data.ContinuousVariable attribute), 30
anonymous (Orange.data.Domain attribute), 24
ANOVA (class in Orange.preprocess.score), 46
append() (Orange.data.Table method), 21
attributes (Orange.data.Domain attribute), 24
attributes (Orange.data.Variable attribute), 28
attributes() (Orange.data.Instance method), 34
AUC, 69
AUC() (in module Orange.evaluation), 69
axis (Orange.misc.distmatrix.DistMatrix attribute), 73

B

base_value (Orange.data.DiscreteVariable attribute), 30
build_tree() (Orange.classification.TreeLearner method), 53
build_tree() (Orange.regression.TreeLearner method), 61

C

CA, 68
CA() (in module Orange.evaluation), 68
case_sensitive (Orange.data.filter.FilterString attribute), 36
case_sensitive (Orange.data.filter.FilterStringList attribute), 37
CD diagram, 70
checksum() (Orange.data.Table method), 21
Chi2 (class in Orange.preprocess.score), 46
class_type (Orange.preprocess.score.ANOVA attribute), 46
class_type (Orange.preprocess.score.Chi2 attribute), 46
class_type (Orange.preprocess.score.FCBF attribute), 47
class_type (Orange.preprocess.score.GainRatio attribute), 46
class_type (Orange.preprocess.score.Gini attribute), 47
class_type (Orange.preprocess.score.InfoGain attribute), 47
class_type (Orange.preprocess.score.ReliefF attribute), 47
class_type (Orange.preprocess.score.RReliefF attribute), 48
class_type (Orange.preprocess.score.UnivariateLinearRegression attribute), 47
class_var (Orange.data.Domain attribute), 24
class_vars (Orange.data.Domain attribute), 24
classes() (Orange.data.Instance method), 34
classification, 8
    accuracy, 10
    area under ROC, 10
    classifier, 9
    elliptic envelope, 54
    k-nearest neighbors, 11, 51
    learner, 9
    linear SVM, 52

logistic regression, 9, 11, 49
majority, 54
naive Bayes, 51
neural network, 54
Nu-SVM, 52
one class SVM, 52
random forest, 49
rules, 55
scoring, 10
simple random forest, 50
simple tree, 54
softmax regression, 50
SVM, 51
tree, 53
trees, 11
classification tree, 53
classification tree (simple), 54
clear() (Orange.data.Table method), 21
clustering
hierarchical clustering, 61
CN2Learner (class in Orange.classification.rules), 55
CN2SDLearner (class in Orange.classification.rules), 56
CN2SDUnorderedLearner (class in Orange.classification.rules), 56
CN2UnorderedLearner (class in Orange.classification.rules), 55
col_items (Orange.misc.distmatrix.DistMatrix attribute), 73
column (Orange.data.filter.FilterContinuous attribute), 36
column (Orange.data.filter.FilterDiscrete attribute), 36
column (Orange.data.filter.FilterString attribute), 36
column (Orange.data.filter.FilterStringList attribute), 37
column (Orange.data.filter.SameValue attribute), 35
column (Orange.data.filter.ValueFilter attribute), 36
columns (Orange.data.filter.IsDefined attribute), 35
columns (Orange.data.Table attribute), 19
compute_CD() (in module Orange.evaluation), 70
compute_value (Orange.data.Variable attribute), 28, 29
conditions (Orange.data.filter.Values attribute), 35
conjunction (Orange.data.filter.Values attribute), 35
connection (Orange.data.sql.table.SqlTable attribute), 22
ContinuousVariable (class in Orange.data), 30
copy() (Orange.data.sql.table.SqlTable method), 23
cross-validation, 10
CrossValidation (class in Orange.evaluation.testing), 67
CrossValidationFeature (class in Orange.evaluation.testing), 67

D

Data, 39
data
attributes, 3
class, 3
domain, 3
examples, 4
input, 1
instances, 4
missing values, 7
preprocessing, 40
sampling, 8
data mining
supervised, 8
database (Orange.data.sql.table.SqlTable attribute), 22
dim (Orange.misc.distmatrix.DistMatrix attribute), 73
DiscreteVariable (class in Orange.data), 30
Discretization (class in Orange.preprocess.discretize), 41
discretize data, 40
Distance (class in Orange.distance), 63
DistMatrix (class in Orange.misc.distmatrix), 73
Domain (class in Orange.data), 24
domain (in module Orange.data.storage), 16
domain (Orange.data.Instance attribute), 33
domain (Orange.data.Table attribute), 19
download_data() (Orange.data.sql.table.SqlTable method), 23

E

elliptic envelope, 54
classification, 54
EllipticEnvelopeLearner (class in Orange.classification), 54
ensure_copy() (Orange.data.Table method), 21
EntropyMDL (class in Orange.preprocess.discretize), 41
EqualFreq (class in Orange.preprocess.discretize), 41
EqualWidth (class in Orange.preprocess.discretize), 41
extend() (Orange.data.Table method), 21

F

F1, 68
F1() (in module Orange.evaluation), 68
FCBF (class in Orange.preprocess.score), 47
feature
discretize, 40
selection, 8
feature (Orange.evaluation.testing.CrossValidationFeature attribute), 67
feature_type (Orange.preprocess.score.ANOVA attribute), 46
feature_type (Orange.preprocess.score.Chi2 attribute), 46
feature_type (Orange.preprocess.score.FCBF attribute), 47
feature_type (Orange.preprocess.score.GainRatio attribute), 46
feature_type (Orange.preprocess.score.Gini attribute), 47
feature_type (Orange.preprocess.score.InfoGain attribute), 47
feature_type (Orange.preprocess.score.ReliefF attribute), 47

feature_type (Orange.preprocess.score.RReliefF attribute), 48
feature_type (Orange.preprocess.score.UnivariateLinearRegression attribute), 47
Filter (class in Orange.data.filter), 35
FilterContinuous (class in Orange.data.filter), 36
FilterDiscrete (class in Orange.data.filter), 36
FilterRegex (class in Orange.data.filter), 37
FilterString (class in Orange.data.filter), 36
FilterStringList (class in Orange.data.filter), 37
fit() (Orange.evaluation.testing.Results method), 66
fit_storage() (Orange.regression.MeanLearner method), 59
force (Orange.preprocess.EntropyMDL attribute), 41
from_domain() (Orange.data.Table class method), 19
from_file() (Orange.data.Table class method), 20
from_file() (Orange.misc.distmatrix.DistMatrix class method), 73
from_numpy() (Orange.data.Domain class method), 25
from_numpy() (Orange.data.Table class method), 20
from_table() (Orange.data.Table class method), 20
from_table_rows() (Orange.data.Table class method), 20

G

GainRatio (class in Orange.preprocess.score), 46
get_augmented_data() (Orange.evaluation.testing.Results method), 66
get_class() (Orange.data.Instance method), 34
get_classes() (Orange.data.Instance method), 34
Gini (class in Orange.preprocess.score), 46
graph_ranks() (in module Orange.evaluation), 70

H

has_col_labels() (Orange.misc.distmatrix.DistMatrix method), 74
has_continuous_attributes() (Orange.data.Domain method), 26
has_discrete_attributes() (Orange.data.Domain method), 26
has_missing() (Orange.data.Table method), 21
has_missing_class() (Orange.data.Table method), 21
has_row_labels() (Orange.misc.distmatrix.DistMatrix method), 74
has_weights() (Orange.data.Table method), 21
HasClass (class in Orange.data.filter), 35
hierarchical clustering, 61
    clustering, 61
HierarchicalClustering (class in Orange.clustering.hierarchical), 62
host (Orange.data.sql.table.SqlTable attribute), 22

I

ids (Orange.data.sql.table.SqlTable attribute), 23
IncrementalPCA (class in Orange.projection.pca), 72
index() (Orange.data.Domain method), 26
InfoGain (class in Orange.preprocess.score), 47
insert() (Orange.data.Table method), 21
Instance (class in Orange.data), 33
is_copy() (Orange.data.Table method), 21
is_primitive() (Orange.data.ContinuousVariable method), 30
is_primitive() (Orange.data.DiscreteVariable method), 31
is_primitive() (Orange.data.StringVariable method), 31
is_primitive() (Orange.data.Variable class method), 28
is_view() (Orange.data.Table method), 21
IsDefined (class in Orange.data.filter), 35

K

k (Orange.evaluation.testing.CrossValidation attribute), 67
k-nearest neighbors
    classification, 51
k-nearest neighbors classifier, 51
KNNLearner (class in Orange.classification), 51

L

LassoRegressionLearner (class in Orange.regression.linear), 57
LeaveOneOut (class in Orange.evaluation.testing), 67
linear, 52
linear fitter
    regression, 57
linear SVM
    classification, 52
LinearModel (class in Orange.regression.linear), 58
LinearRegressionLearner (class in Orange.regression.linear), 57
LinearSVMLearner (class in Orange.classification), 52
list (Orange.data.Instance attribute), 34
Log loss, 69
logistic regression, 49
    classification, 49
LogisticRegressionLearner (class in Orange.classification), 49
LogLoss() (in module Orange.evaluation), 69

M

MAE, 70
MAE() (in module Orange.evaluation), 70
majority
    classification, 54
majority classifier, 54
MajorityLearner (class in Orange.classification), 54
make() (Orange.data.ContinuousVariable method), 30
make() (Orange.data.DiscreteVariable class method), 30
make() (Orange.data.StringVariable method), 31
master (Orange.data.Variable attribute), 28
max (Orange.data.filter.FilterContinuous attribute), 36

max (Orange.data.filter.FilterString attribute), 36
mean fitter, 58
    regression, 58
MeanLearner (class in Orange.regression), 59
metas (Orange.data.Domain attribute), 24
metas (Orange.data.Instance attribute), 34
metas (Orange.data.sql.table.SqlTable attribute), 23
MSE, 70
MSE() (in module Orange.evaluation), 70
multinomial_treatment (Orange.preprocess.Orange.preprocess.Continuize attribute), 42

N

n (Orange.preprocess.EqualFreq attribute), 41
n (Orange.preprocess.EqualWidth attribute), 41
naive Bayes
    classification, 51
naive Bayes classifier, 51
NaiveBayesLearner (class in Orange.classification), 51
name (Orange.data.Variable attribute), 28
negate (Orange.data.filter.Filter attribute), 35
negate (Orange.data.filter.Values attribute), 35
neural network, 54, 61
    classification, 54
    regression, 61
NNClassificationLearner (class in Orange.classification), 55
NNRegressionLearner (class in Orange.regression), 61
Normalize (class in Orange.preprocess), 44
Nu-SVM, 52
    classification, 52
number_of_decimals (Orange.data.ContinuousVariable attribute), 30
NuSVMLearner (class in Orange.classification), 52

O

one class SVM, 52
    classification, 52
OneClassSVMLearner (class in Orange.classification), 53
oper (Orange.data.filter.FilterContinuous attribute), 36
oper (Orange.data.filter.FilterString attribute), 36
Orange.classification (module), 49
Orange.classification.rules (module), 55
Orange.clustering (module), 61
Orange.data.filter (module), 35
Orange.data.variable (module), 32
Orange.evaluation (module), 66
Orange.evaluation.testing (module), 66
Orange.misc (module), 73
Orange.misc.distmatrix (module), 73
Orange.preprocess.Continuize (class in Orange.preprocess), 41
Orange.preprocess.DomainContinuizer (class in Orange.preprocess), 43
Orange.projection (module), 71
Orange.regression (module), 57
ordered (Orange.data.DiscreteVariable attribute), 30

P

parse() (Orange.data.TimeVariable method), 32
PCA (class in Orange.projection.pca), 72
PolynomialLearner (class in Orange.regression.linear), 58
Precision, 68
Precision() (in module Orange.evaluation), 68
PrecisionRecallFSupport, 69
PrecisionRecallFSupport() (in module Orange.evaluation), 69
prepare_arrays() (Orange.evaluation.testing.Results method), 66
preprocessing, 40
prob (Orange.data.filter.Random attribute), 35

R

R2, 70
R2() (in module Orange.evaluation), 70
Random (class in Orange.data.filter), 35
random forest, 49, 59
    classification, 49
    regression, 59
random forest (simple), 50, 59
random_state (Orange.evaluation.testing.CrossValidation attribute), 67
RandomForestLearner (class in Orange.classification), 49
RandomForestRegressionLearner (class in Orange.regression), 59
Randomize (class in Orange.preprocess), 44
Recall, 68
Recall() (in module Orange.evaluation), 68
ref (Orange.data.filter.FilterContinuous attribute), 36
ref (Orange.data.filter.FilterString attribute), 36
regression, 12
    linear, 12
    linear fitter, 57
    mean fitter, 58
    neural network, 61
    random forest, 59
    simple random forest, 59
    tree, 12, 60
regression tree, 60
ReliefF (class in Orange.preprocess.score), 47
Remove (class in Orange.preprocess), 45
Results (class in Orange.evaluation.testing), 66
RidgeRegressionLearner (class in Orange.regression.linear), 57
row_filters (Orange.data.sql.table.SqlTable attribute), 22

row_items (Orange.misc.distmatrix.DistMatrix attribute), 73
RowInstance (class in Orange.data), 34
RReliefF (class in Orange.preprocess.score), 48
Rule induction, 55
rules
    classification, 55

S

SameValue (class in Orange.data.filter), 35
sample() (in module Orange.evaluation.testing), 67
save() (Orange.misc.distmatrix.DistMatrix method), 74
SelectBestFeatures (class in Orange.preprocess), 48
set_class() (Orange.data.Instance method), 34
set_weights() (Orange.data.Table method), 21
setup_indices() (Orange.evaluation.testing.Results method), 66
SGDRegressionLearner (class in Orange.regression.linear), 58
shuffle() (Orange.data.Table method), 21
simple random forest
    classification, 50
    regression, 59
simple tree
    classification, 54
SimpleRandomForestLearner (class in Orange.classification), 50
SimpleRandomForestLearner (class in Orange.regression), 59
SimpleTreeLearner (class in Orange.classification), 54
SklTreeLearner (class in Orange.classification), 53
SklTreeRegressionLearner (class in Orange.regression), 61
softmax regression
    classification, 50
softmax regression classifier, 50
SoftmaxRegressionLearner (class in Orange.classification), 50
source_variable (Orange.data.Variable attribute), 28
SparsePCA (class in Orange.projection.pca), 72
split_by_model() (Orange.evaluation.testing.Results method), 67
SqlRowInstance (class in Orange.data.sql.table), 23
SqlTable (class in Orange.data.sql.table), 22
str_val() (Orange.data.ContinuousVariable method), 30
str_val() (Orange.data.DiscreteVariable method), 31
str_val() (Orange.data.StringVariable static method), 31
str_val() (Orange.data.Variable method), 28
StringVariable (class in Orange.data), 31
submatrix() (Orange.misc.distmatrix.DistMatrix method), 73
SVM, 51, 52
    classification, 51
SVMLearner (class in Orange.classification), 52

T

Table (class in Orange.data), 18
table_name (Orange.data.sql.table.SqlTable attribute), 22
TestOnTestData (class in Orange.evaluation.testing), 67
TestOnTrainingData (class in Orange.evaluation.testing), 67
TimeVariable (class in Orange.data), 32
to_val() (Orange.data.ContinuousVariable method), 30
to_val() (Orange.data.DiscreteVariable method), 31
to_val() (Orange.data.StringVariable method), 31
to_val() (Orange.data.Variable method), 28
total_weight() (Orange.data.Table method), 21
transform_class (Orange.preprocess.Orange.preprocess.Continuize attribute), 43
tree
    classification, 53
    regression, 60
TreeLearner (class in Orange.classification), 53
TreeLearner (class in Orange.regression), 60
Type (Orange.data.filter.FilterContinuous attribute), 36
Type (Orange.data.filter.FilterString attribute), 37

U

UnivariateLinearRegression (class in Orange.preprocess.score), 47
unknown_str (Orange.data.Variable attribute), 28

V

val_from_str_add() (Orange.data.ContinuousVariable method), 30
val_from_str_add() (Orange.data.DiscreteVariable method), 31
val_from_str_add() (Orange.data.StringVariable method), 31
val_from_str_add() (Orange.data.Variable method), 29
Value (class in Orange.data.variable), 32
value (Orange.data.filter.SameValue attribute), 35
value (Orange.data.variable.Value attribute), 32
ValueFilter (class in Orange.data.filter), 36
Values (class in Orange.data.filter), 35
values (Orange.data.DiscreteVariable attribute), 30
values (Orange.data.filter.FilterDiscrete attribute), 36
values (Orange.data.filter.FilterStringList attribute), 37
Variable (class in Orange.data), 28
variable (Orange.data.variable.Value attribute), 32
variables (Orange.data.Domain attribute), 24

W

W (Orange.data.sql.table.SqlTable attribute), 23
weight (Orange.data.Instance attribute), 34

X

x (Orange.data.Instance attribute), 34

X (Orange.data.sql.table.SqlTable attribute), 23

Y

y (Orange.data.Instance attribute), 34
Y (Orange.data.sql.table.SqlTable attribute), 23

Z

zero_based (Orange.preprocess.Orange.preprocess.Continuize attribute), 41