Orange Data Mining Library Documentation

Release 3

Orange Data Mining

Sep 24, 2021

Contents

1 Tutorial **1**

1.1 The Data . . . 1
 1.1.1 Data Input . . . 1
 1.1.2 Creating a Data Table . . . 3
 1.1.3 Saving the Data . . . 4
 1.1.4 Exploration of the Data Domain . . . 4
 1.1.5 Data Instances . . . 5
 1.1.6 Orange Datasets and NumPy . . . 6
 1.1.7 Meta Attributes . . . 7
 1.1.8 Missing Values . . . 8
 1.1.9 Data Selection and Sampling . . . 9
1.2 Classification . . . 10
 1.2.1 Learners and Classifiers . . . 10
 1.2.2 Probabilistic Classification . . . 11
 1.2.3 Cross-Validation . . . 11
 1.2.4 Handful of Classifiers . . . 12
1.3 Regression . . . 13
 1.3.1 Handful of Regressors . . . 13
 1.3.2 Cross Validation . . . 14

2 Reference **17**

2.1 Data model (`data`) . . . 17
 2.1.1 Data Storage (`storage`) . . . 18
 2.1.2 Data Table (`table`) . . . 20
 2.1.3 SQL table (`data.sql`) . . . 20
 2.1.4 Domain description (`domain`) . . . 20
 2.1.5 Variable Descriptors (`variable`) . . . 21
 2.1.6 Values (`value`) . . . 25
 2.1.7 Data Instance (`instance`) . . . 25
 2.1.8 Data Filters (`filter`) . . . 26
 2.1.9 Loading and saving data (`io`) . . . 26
2.2 Data Preprocessing (`preprocess`) . . . 28
 2.2.1 Impute . . . 28
 2.2.2 Discretization . . . 28
 2.2.3 Continuization . . . 29
 2.2.4 Normalization . . . 31

2.2.5 Randomization . . . 31
2.2.6 Remove . . . 31
2.2.7 Feature selection . . . 31
2.2.8 Preprocessors . . . 33
2.3 Outlier detection (`classification`) . . . 33
2.3.1 One Class Support Vector Machines . . . 33
2.3.2 Elliptic Envelope . . . 33
2.3.3 Local Outlier Factor . . . 33
2.3.4 Isolation Forest . . . 33
2.4 Classification (`classification`) . . . 33
2.4.1 Logistic Regression . . . 33
2.4.2 Random Forest . . . 33
2.4.3 Simple Random Forest . . . 33
2.4.4 Softmax Regression . . . 33
2.4.5 k-Nearest Neighbors . . . 33
2.4.6 Naive Bayes . . . 33
2.4.7 Support Vector Machines . . . 33
2.4.8 Linear Support Vector Machines . . . 33
2.4.9 Nu-Support Vector Machines . . . 33
2.4.10 Classification Tree . . . 33
2.4.11 Simple Tree . . . 34
2.4.12 Majority Classifier . . . 34
2.4.13 Neural Network . . . 34
2.4.14 CN2 Rule Induction . . . 34
2.4.15 Calibration and threshold optimization . . . 34
2.4.16 Gradient Boosted Trees . . . 34
2.5 Regression (`regression`) . . . 34
2.5.1 Linear Regression . . . 34
2.5.2 Polynomial . . . 35
2.5.3 Mean . . . 35
2.5.4 Random Forest . . . 35
2.5.5 Simple Random Forest . . . 35
2.5.6 Regression Tree . . . 35
2.5.7 Neural Network . . . 35
2.5.8 Gradient Boosted Trees . . . 35
2.5.9 Curve Fit . . . 35
2.6 Clustering (`clustering`) . . . 35
2.6.1 Hierarchical (`hierarchical`) . . . 35
2.7 Distance (`distance`) . . . 36
2.7.1 Handling discrete and missing data . . . 37
2.7.2 Supported distances . . . 37
2.8 Evaluation (`evaluation`) . . . 39
2.8.1 Sampling procedures for testing models (`testing`) . . . 39
2.8.2 Scoring methods (`scoring`) . . . 39
2.8.3 Performance curves . . . 40
2.9 Projection (`projection`) . . . 40
2.9.1 PCA . . . 40
2.9.2 FreeViz . . . 40
2.9.3 LDA . . . 41
2.9.4 References . . . 42
2.10 Miscellaneous (`misc`) . . . 42
2.10.1 Distance Matrix (`distmatrix`) . . . 42

Index **43**

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
'3.25.0.dev0+3bdef92'
>>>
```

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

1.1 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.

1.1.1 Data Input

Orange can read files in proprietary tab-delimited format, or can load data from any of the major standard spreadsheet file types, like CSV and Excel. Native format starts with a header row with feature (column) names. The second header row gives the attribute type, which can be numeric, categorical, 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 dataset `lenses.tab`:

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

(continues on next page)

(continued from previous page)

```
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 dataset 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 dataset 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 dataset, check out or download `lenses.tab`) to a target directory. You can also skip this step as Orange comes preloaded with several demo datasets, 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 a readable type. The call to `Orange.data.Table` creates an object called `data` that holds your dataset 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` prescription:

```
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.

1.1.2 Creating a Data Table

To create a data table from scratch, one needs two things, a domain and the data. The domain is the description of the variables, i.e. column names, types, roles, etc.

First, we create the said domain. We will create three types of variables, numeric (ContiniousVariable), categorical (DiscreteVariable) and text (StringVariable). Numeric and categorical variables will be used a features (also known as X), while the text variable will be used as a meta variable.

```
>>> from Orange.data import Domain, ContinuousVariable,
    DiscreteVariable, StringVariable
>>>
>>> domain = Domain([ContinuousVariable("col1"),
                     DiscreteVariable("col2", values=["red", "blue"])],
                     metas=[StringVariable("col3")])
```

Now, we will build the data with numpy.

```
>>> import numpy as np
>>>
>>> column1 = np.array([1.2, 1.4, 1.5, 1.1, 1.2])
>>> column2 = np.array([0, 1, 1, 1, 0])
>>> column3 = np.array(["U13", "U14", "U15", "U16", "U17"], dtype=object)
```

Two things to note here. column2 has values 0 and 1, even though we specified it will be a categorical variable with values "red" and "blue". X (features in the data) can only be numbers, so the numpy matrix will contain numbers, while Orange will handle the categorical representation internally. 0 will be mapped to the value "red" and 1 to "blue" (in the order, specified in the domain).

Text variable requires `dtype=object` for numpy to handle it correctly.

Next, variables have to be transformed to a matrix.

```
>>> X = np.column_stack((column1, column2))
>>> M = column3.reshape(-1, 1)
```

Finally, we create a table. We need a domain and variables, which can be passed as X (features), Y (class variable) or metas.

```
>>> table = Table.from_numpy(domain, X=X, metas=M)
>>> print(table)
>>> [[1.2, red] {U13},
[1.4, blue] {U14},
[1.5, blue] {U15},
[1.1, blue] {U16},
[1.2, red] {U17}]
```

To add a class variable to the table, the procedure would be the same, with the class variable passed as Y (e.g. `table = Table.from_numpy(domain, X=X, Y=Y, metas=M)`).

To add a single column to the table, one can use the `Table.add_column()` method.

```
>>> new_var = DiscreteVariable("var4", values=["one", "two"])
>>> var4 = np.array([0, 1, 0, 0, 1]) # no reshaping necessary
>>> table = table.add_column(new_var, var4)
>>> print(table)
>>> [[1.2, red, one] {U13},
[1.4, blue, two] {U14},
[1.5, blue, one] {U15},
[1.1, blue, one] {U16},
[1.2, red, two] {U17}]
```

1.1.3 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.

1.1.4 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
```

1.1.5 Data Instances

Data table stores data instances (or examples). These can be indexed 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
```

The Iris dataset we have used above has four continuous 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])))
```

The above script 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
```

(continues on next page)

(continued from previous page)

```
sepal width     3.05
petal length    3.76
petal width     1.20
```

A slightly more complicated, but also more interesting, 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" % ("Feature", " ".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 dataset:

```
import Orange
from collections import Counter

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

1.1.6 Orange Datasets and NumPy

Orange datasets 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 dataset:

```
>>> 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) dataset 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 the construction of a dataset that includes a numerical class and different types 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]
```

1.1.7 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 separate numpy arrays:

```
>>> data = Orange.data.Table("zoo")
>>> data[0]["name"]
```

(continues on next page)

(continued from previous page)

```
>>> 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.

1.1.8 Missing Values

Consider the following exploration of the dataset 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 dataset 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" % (100.0 * 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
```

1.1.9 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 dataset. This is useful for any data subsetting:

```
data = Orange.data.Table("iris.tab")
print("Dataset 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:

```
Dataset instances: 150
Subset size: 99
```

and inherits the data description (domain) from the original dataset. 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 dataset:

```
>>> 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]
```

1.2 Classification

Much of Orange is devoted to machine learning methods for classification, or supervised data mining. These methods rely on data with class-labeled instances, like that of senate voting. Here is a code that loads this dataset, 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 functions for construction of classification models, their evaluation and scoring. In a nutshell, 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 receive the data and return classifiers. Learners are passed to evaluation routines, such as cross-validation above.

1.2.1 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 dataset 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 dataset:

```
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))], 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 dataset:

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

1.2.2 Probabilistic Classification

To find out what is the probability that the classifier assigns to, say, democrat class, we need to call the classifier with an 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
```

1.2.3 Cross-Validation

Validating the accuracy of classifiers on the training data, as we did above, serves demonstration purposes only. Any performance measure that assesses 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 dataset:

```
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 an array of length one was returned. The script estimates classification accuracy and area under ROC curve:

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

1.2.4 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`)
- random 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 datasets 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)[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 dataset.

```
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
```

1.3 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), d.get_class()))
```

1.3.1 Handful of Regressors

Let us start with regression trees. Below is an example script that builds a 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 the initialization of a few other regressors and their prediction of the first five data instances in the housing price dataset:

```

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) 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
```

1.3.2 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 with two measures for goodness 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 the list of our regressors; this regressor simply predicts the mean value from the training set, and is used as a baseline.

CHAPTER 2

Reference

Available classes and methods.

2.1 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.

2.1.1 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

2.1.2 Data Table (`table`)

Constructors

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

Inspection

Row manipulation

Note: Methods that change the table length (*append*, *extend*, *insert*, *clear*, and resizing through deleting, slicing or by other means), were deprecated and removed in Orange 3.24.

Weights

2.1.3 SQL table (`data.sql`)

2.1.4 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]
```

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.

2.1.5 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 datasets are considered to be the same if they have the same descriptor;
- conversion of values between domains or datasets, 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')
```

(continues on next page)

(continued from previous page)

```
>>> 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 *Derived variables* for a detailed explanation.

Constructors

Orange maintains lists of existing descriptors for variables. This facilitates the reuse of descriptors: if two datasets refer to the same variables, they should be assigned the same descriptors so that, for instance, a model trained on one dataset 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

Continuous variables

Discrete variables

String variables

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.

Derived variables

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

Method *compute_value* is usually invoked behind the scenes in conversion of domains. Such conversions are are typically implemented within the provided wrappers and cross-validation schemes.

Derived variables in Orange

Orange saves variable transformations into the domain as `compute_value` functions. If Orange was not using `compute_value`, we would have to manually transform the data:

```
>>> from Orange.data import Domain, ContinuousVariable
>>> data = Orange.data.Table("iris")
>>> train = data[::2]   # every second row
>>> test = data[1::2]   # every other second instance
```

We will create a new data set with a single feature, "petals", that will be a sum of petal lengths and widths:

```
>>> petals = ContinuousVariable("petals")
>>> derived_train = train.transform(Domain([petals],
...                                 data.domain.class_vars))
>>> derived_train.X = train[:, "petal width"].X + \
...                   train[:, "petal length"].X
```

We have set `Table`'s *X* directly. Next, we build and evaluate a classification tree:

```
>>> learner = Orange.classification.TreeLearner()
>>> from Orange.evaluation import CrossValidation, TestOnTestData
>>> res = CrossValidation(derived_train, [learner], k=5)
>>> Orange.evaluation.scoring.CA(res)[0]
0.88
>>> res = TestOnTestData(derived_train, test, [learner])
>>> Orange.evaluation.scoring.CA(res)[0]
0.3333333333333333
```

A classification tree shows good accuracy with cross validation, but not on separate test data, because Orange can not reconstruct the "petals" feature for test data—we would have to reconstruct it ourselves. But if we define `compute_value` and therefore store the transformation in the domain, Orange could transform both training and test data:

```
>>> petals = ContinuousVariable("petals",
...     compute_value=lambda data: data[:, "petal width"].X + \
...                               data[:, "petal length"].X)
>>> derived_train = train.transform(Domain([petals],
...                                 data.domain.class_vars))
>>> res = TestOnTestData(derived_train, test, [learner])
```

(continues on next page)

(continued from previous page)

```
>>> Orange.evaluation.scoring.CA(res)[0]
0.9733333333333334
```

All preprocessors in Orange use `compute_value`.

Example with discretization

The following example converts features to discrete:

```
>>> iris = Orange.data.Table("iris")
>>> iris_1 = iris[::2]
>>> discretizer = Orange.preprocess.DomainDiscretizer()
>>> d_iris_1 = discretizer(iris_1)
```

A dataset is loaded and a new table with every second instance is created. On this dataset, we compute discretized data, which uses the same data to set proper discretization intervals.

The discretized variable "D_sepal length" stores a function that can derive continous values into discrete:

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

The function is used for converting the remaining data (as automatically happens within model validation in Orange):

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

The code transforms previously unused data into the discrete domain *d_iris_1.domain*. Behind the scenes, the values for the destination domain that are not yet in the source domain (*iris_2.domain*) are computed with the destination variables' `compute_value`.

Optimization for repeated computation

Some transformations share parts of computation across variables. For example, `PCA` uses all input features to compute the PCA transform. If each output PCA component was implemented with ordinary `compute_value`, the PCA transform would be repeatedly computed for each PCA component. To avoid repeated computation, set `compute_value` to a subclass of `SharedComputeValue`.

The following example creates normalized features that divide values by row sums and then tranforms the data. In the example the function *row_sum* is called only once; if we did not use `SharedComputeValue`, *row_sum* would be called four times, once for each feature.

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

def row_sum(data):
    return data.X.sum(axis=1, keepdims=True)

class DivideWithMean(Orange.data.util.SharedComputeValue):

```

(continues on next page)

(continued from previous page)

```
    def __init__(self, var, fn):
        super().__init__(fn)
        self.var = var

    def compute(self, data, shared_data):
        return data[:, self.var].X / shared_data

divided_attributes = [
    Orange.data.ContinuousVariable(
        "Divided " + attr.name,
        compute_value=DivideWithMean(attr, row_sum)
    ) for attr in iris.domain.attributes]

divided_domain = Orange.data.Domain(
    divided_attributes,
    iris.domain.class_vars
)

divided_iris = iris.transform(divided_domain)
```

2.1.6 Values (`value`)

2.1.7 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 dataset.

```
>>> 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
```

Rows of Data Tables

2.1.8 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]*.

2.1.9 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),
- 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,

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>` are custom attributes recognized in specific contexts, for instance `color`, which defines the color palette when the variable is visualized, or `type=image` which signals that the variable contains a path to an image.

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 (also known as a target variable or dependent variable),
- `i` for feature to be ignored,
- `m` for meta attributes (not used in learning),
- `C` for features that are continuous (numeric),
- `D` for features that are discrete (categorical),
- `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).

2.2 Data Preprocessing (`preprocess`)

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

2.2.1 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)
```

2.2.2 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 dataset:")
for e in iris[:3]:
    print(e)

print("Discretized dataset:")
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 dataset:
[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 dataset:
[<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.

```
iris = Orange.data.Table("iris.tab")
disc = Orange.preprocess.Discretize()
disc.method = Orange.preprocess.discretize.EqualFreq(n=2)
```

Discretization Algorithms

To add a new discretization, derive it from `Discretization`.

2.2.3 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, dataset "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.

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 dataset 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.

2.2.4 Normalization

2.2.5 Randomization

2.2.6 Remove

2.2.7 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 dataset:

```
>>> data = Orange.data.Table("lenses")
>>> Orange.preprocess.score.InfoGain(data, "tear_rate")
0.548794940695398580
```

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.

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 dataset.

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 dataset 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]
```

2.2.8 Preprocessors

2.3 Outlier detection (`classification`)

2.3.1 One Class Support Vector Machines

2.3.2 Elliptic Envelope

2.3.3 Local Outlier Factor

2.3.4 Isolation Forest

2.4 Classification (`classification`)

2.4.1 Logistic Regression

2.4.2 Random Forest

2.4.3 Simple Random Forest

2.4.4 Softmax Regression

2.4.5 k-Nearest Neighbors

2.4.6 Naive Bayes

The following code loads lenses dataset (four discrete attributes and discrete class), constructs naive Bayesian learner, uses it on the entire dataset 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]])
```

2.4.7 Support Vector Machines

2.4.8 Linear Support Vector Machines

2.4.9 Nu-Support Vector Machines

2.4.10 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 still faster.

The following code loads iris dataset (four numeric attributes and discrete class), constructs a decision tree learner, uses it on the entire dataset to construct a classifier, and then prints the tree:

```
>>> import Orange
>>> iris = Orange.data.Table('iris')
>>> tr = Orange.classification.TreeLearner()
>>> classifier = tr(data)
>>> printed_tree = classifier.print_tree()
>>> for i in printed_tree.split('\n'):
>>>     print(i)
[50.  0.  0.] petal length  1.9
[ 0. 50. 50.] petal length > 1.9
[ 0. 49.  5.]     petal width  1.7
[ 0. 47.  1.]         petal length  4.9
   [0. 2. 4.]         petal length > 4.9
   [0. 0. 3.]             petal width  1.5
   [0. 2. 1.]             petal width > 1.5
   [0. 2. 0.]                 sepal length  6.7
   [0. 0. 1.]                 sepal length > 6.7
[ 0.  1. 45.]     petal width > 1.7
```

2.4.11 Simple Tree

2.4.12 Majority Classifier

2.4.13 Neural Network

2.4.14 CN2 Rule Induction

2.4.15 Calibration and threshold optimization

2.4.16 Gradient Boosted Trees

2.5 Regression (`regression`)

2.5.1 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)
```

(continues on next page)

(continued from previous page)

```
>>> model(mpg[0])
Value('mpg', 24.6)
```

2.5.2 Polynomial

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

2.5.3 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])
```

2.5.4 Random Forest

2.5.5 Simple Random Forest

2.5.6 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.

2.5.7 Neural Network

2.5.8 Gradient Boosted Trees

2.5.9 Curve Fit

2.6 Clustering (`clustering`)

2.6.1 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.,  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.,  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

2.7 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.

2.7.1 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.

2.7.2 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.

2.8 Evaluation (`evaluation`)

2.8.1 Sampling procedures for testing models (`testing`)

2.8.2 Scoring methods (`scoring`)

CA

Precision

Recall

F1

PrecisionRecallFSupport

AUC

Log Loss

MSE

MAE

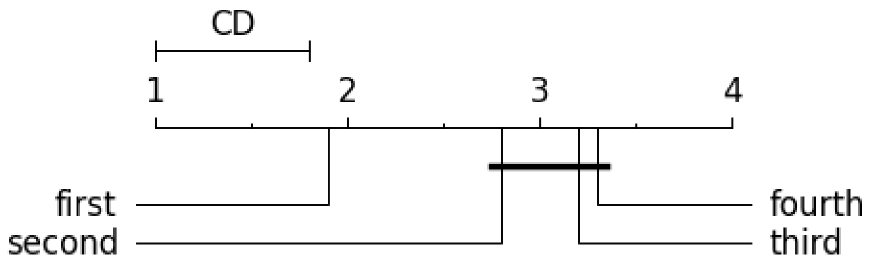
R2

CD diagram

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:



2.8.3 Performance curves

2.9 Projection (`projection`)

2.9.1 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 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],
...
]
```

2.9.2 FreeViz

FreeViz uses a paradigm borrowed from particle physics: points in the same class attract each other, those from different class repel each other, and the resulting forces are exerted on the anchors of the attributes, that is, on unit vectors of each of the dimensional axis. The points cannot move (are projected in the projection space), but the attribute anchors can, so the optimization process is a hill-climbing optimization where at the end the anchors are placed such that forces are in equilibrium.

Example

```
>>> from Orange.projection import FreeViz
>>> from Orange.data import Table
>>> iris = Table('iris')
>>> freeviz = FreeViz()
>>> model = freeviz(iris)
>>> model.components_    # FreeViz components
array([[  3.83487853e-01,   1.38777878e-17],
   [ -6.95058218e-01,   7.18953457e-01],
   [  2.16525357e-01,  -2.65741729e-01],
   [  9.50450079e-02,  -4.53211728e-01]])
>>> transformed_data = model(iris)    # transformed data
>>> transformed_data
[[-0.157, 2.053 | Iris-setosa],
[0.114, 1.694 | Iris-setosa],
[-0.123, 1.864 | Iris-setosa],
[-0.048, 1.740 | Iris-setosa],
[-0.265, 2.125 | Iris-setosa],
...
]
```

2.9.3 LDA

Linear discriminant analysis is another way of finding a linear transformation of data that reduces the number of dimensions required to represent it. It is often used for dimensionality reduction prior to classification, but can also be used as a classification technique itself ([1]).

Example

```
>>> from Orange.projection import LDA
>>> from Orange.data import Table
>>> iris = Table('iris')
>>> lda = LDA()
>>> model = LDA(iris)
>>> model.components_    # LDA components
array([[ 0.20490976,  0.38714331, -0.54648218, -0.71378517],
   [ 0.00898234,  0.58899857, -0.25428655,  0.76703217],
   [-0.71507172,  0.43568045,  0.45568731, -0.30200008],
   [ 0.06449913, -0.35780501, -0.42514529,  0.828895  ]])
>>> transformed_data = model(iris)    # transformed data
>>> transformed_data
[[1.492, 1.905 | Iris-setosa],
[1.258, 1.608 | Iris-setosa],
[1.349, 1.750 | Iris-setosa],
[1.180, 1.639 | Iris-setosa],
[1.510, 1.963 | Iris-setosa],
...
]
```

[1] Witten, I.H., Frank, E., Hall, M.A. and Pal, C.J., 2016. Data Mining: Practical machine learning tools and techniques. Morgan Kaufmann.

2.9.4 References

2.10 Miscellaneous (`misc`)

2.10.1 Distance Matrix (`distmatrix`)

Index

Symbols

.. index:: linear fitter, 34
__getitem__() (*in module Orange.data.storage*), 18
__len__() (*Orange.data.storage. method*), 18
_compute_distributions() (*in module Orange.data.storage*), 20
_filter_has_class() (*in module Orange.data.storage*), 19
_filter_is_defined() (*in module Orange.data.storage*), 19
_filter_same_value() (*in module Orange.data.storage*), 19
_filter_values() (*in module Orange.data.storage*), 19

A

AUC, 39

C

CA, 39
CD diagram, 39
classification, 10
    accuracy, 12
    area under ROC, 12
    classifier, 10
    elliptic envelope, 33
    isolation forest, 33
    k-nearest neighbors, 12, 33
    learner, 10
    linear SVM, 33
    local outlier factor, 33
    logistic regression, 10, 12, 33
    majority, 34
    naive Bayes, 33
    neural network, 34
    Nu-SVM, 33
    one class SVM, 33
    random forest, 33
    rules, 34
    scoring, 12
    simple random forest, 33
    simple tree, 34
    softmax regression, 33
    SVM, 33
    tree, 33
    trees, 12
classification tree, 33
classification tree (*simple*), 34
clustering
    hierarchical clustering, 35
cross-validation, 11

D

Data, 27
data
    attributes, 4
    class, 4
    domain, 4
    examples, 5
    input, 1
    instances, 5
    missing values, 8
    preprocessing, 28
    sampling, 9
data mining
    supervised, 10
discretize data, 28
domain (*in module Orange.data.storage*), 18

E

elliptic envelope, 33
    classification, 33

F

F1, 39
feature
    discretize, 28
    selection, 9

H

hierarchical clustering, 35
    clustering, 35

I

isolation forest, 33
    classification, 33

K

k-nearest neighbors
    classification, 33
k-nearest neighbors classifier, 33

L

linear, 33
linear fitter
    regression, 34
linear SVM
    classification, 33
local outlier factor, 33
    classification, 33
Log loss, 39
logistic regression, 33
    classification, 33

M

MAE, 39
majority
    classification, 34
majority classifier, 34
mean fitter, 35
    regression, 35
MSE, 39
multinomial_treatment (*Orange.preprocess.Orange.preprocess.Continuize attribute*), 29

N

naive Bayes
    classification, 33
naive Bayes classifier, 33
neural network, 34, 35
    classification, 34
    regression, 35
Nu-SVM, 33
    classification, 33

O

one class SVM, 33
    classification, 33
Orange.preprocess.Continuize (*class in Orange.preprocess*), 29
Orange.preprocess.DomainContinuizer (*class in Orange.preprocess*), 31

P

Precision, 39
PrecisionRecallFSupport, 39
preprocessing, 28

R

R2, 39
random forest, 33, 35
    classification, 33
    regression, 35
random forest (*simple*), 33, 35
Recall, 39
regression, 13
    linear, 14
    linear fitter, 34
    mean fitter, 35
    neural network, 35
    random forest, 35
    simple random forest, 35
    tree, 13, 35
regression tree, 35
Rule induction, 34
rules
    classification, 34

S

simple random forest
    classification, 33
    regression, 35
simple tree
    classification, 34
softmax regression
    classification, 33
softmax regression classifier, 33
SVM, 33
    classification, 33

T

transform_class (*Orange.preprocess.Orange.preprocess.Continuize attribute*), 31
tree
    classification, 33
    regression, 35

Z

zero_based (*Orange.preprocess.Orange.preprocess.Continuize attribute*), 29