OpenML

The OpenML R Team

2016-11-11

Introduction

The R package OpenML is an interface to make interactions with the OpenML server as comfortable as possible. For example, the users can download and upload files, run their implementations on specific tasks and get predictions in the correct form directly via R commands. In this tutorial, we will show the most important functions of this package and give examples on standard workflows.

For general information on what OpenML is, please have a look at the README file or visit the official OpenML website.

After installation and before making practical use of the package, in most cases it is desirable to setup a configuration file to simplify further steps. Afterwards, there are different basic stages when using this package or OpenML, respectively:

- Listing
  - lists which data is available w.r.t. a specific object (`DataSets`, `Tasks`, `Flows`, `Runs`, `RunEvaluations`, `EvaluationMeasures`, and `TaskTypes`)
  - function names begin with `listOML`
  - result is always a `data.frame`
- Downloading
  - downloads the specific objects (for `DataSets`, `Tasks`, `Runs`, `Predictions`, and `Flows`)
  - function names begin with `getOML`
  - result is an object of a specific OpenML class
- Running models on tasks
  - function `runTaskMlr`
  - input: `OMLTask` and `Learner`
  - output: `OMLMlrRun`, `OMLRun`
- Uploading
  - function `uploadOMLRun`

Installation instructions

Installation works as in any other package using

```
install.packages("OpenML")
```

To install the current development version use the `devtools` package and run

```
devtools::install_github("openml/openml-r")
```

Using the OpenML package also requires a reader for the ARFF file format. By default farff is used. Alternatively, the RWeka package can be used. You can install the packages with the following calls.

```
install.packages(c("farff", "RWeka"))
```

Private key notification

All examples in this tutorial are given with a **READ-ONLY API key**.

With this key you can **read** all the information from the server but not **write** data sets, tasks, flows, and runs to the server. This key allows to emulate uploading to the server but doesn't allow to really store data. If one wants to write data to a server, one has to **get a personal API key**. The process of how to obtain a key is shown in the configuration section.

Important: Please do not write meaningless data to the server such as copies of already existing data sets, tasks, or runs (such as the ones from this tutorial)! One instance of the Iris data set should be enough for everyone. :D

Basic example

In this paragraph you can find an example on how to download a task from the server, print some information about it to the console, and produce a run which is then uploaded to the server. For detailed information on OpenML terminology (task, run, etc.) see the OpenML guide.

```
library("OpenML")
## temporarily set API key to read only key
setOMLConfig(apikey = "c1994bdb7ecb3c6f3c8f3b35f4b47f1f", server = "http://test.openml.org/api/v1")
```

```
## OpenML configuration:
##   server           : http://test.openml.org/api/v1
##   cachedir         : L:\GitRepository\openml-r/tests/cache
##   verbosity        : 0
##   arff.reader      : RWeka
##   confirm.upload   : FALSE
##   apikey           : ****************************47f1f
```

```
# download a task (whose ID is 1L)
task = getOMLTask(task.id = 1L)
task
```

```
##
## OpenML Task 1 :: (Data ID = 1)
##   Task Type            : Supervised Classification
##   Data Set             : anneal :: (Version = 2, OpenML ID = 1)
##   Target Feature(s)    : class
##   Tags                 : basic, study_1, study_7, under100k, under1m
##   Estimation Procedure : Stratified crossvalidation (1 x 10 folds)
##   Evaluation Measure(s): predictive_accuracy
```

The task contains information on the following:

- task type: defines the type of the task (regression, classification, clustering, etc.)
- data set: which data set belongs to the given task (one task can always only be connected to a single data set)
- target feature(s): optional field for all kinds of classification and regression tasks
- tags: tags / labels, which might be helpful for further sub-selections
- estimation procedure: which estimation procedure has been used when computing the performance

In the next line, `randomForest` is used as a classifier and run with the help of the `mlr package`. Note that one needs to run the algorithm locally and that `mlr` will automatically download the package that is needed to run the specified classifier.

```
# define the classifier (usually called "flow" within OpenML)
library("mlr")
lrn = makeLearner("classif.randomForest")
```

```
# upload the new flow (with information about the algorithm and settings);
# if this algorithm already exists on the server, one will receive a message
# with the ID of the existing flow
flow.id = uploadOMLFlow(lrn)

# the last step is to perform a run and upload the results
run.mlr = runTaskMlr(task, lrn)
run.id = uploadOMLRun(run.mlr)
```

Following this very brief example, we will explain the single steps of the OpenML package in more detail in the next sections.

Configuration

Generating your own personal API key

The first step of working with OpenML should be to register yourself at the OpenML website. Most of the package's functions require an API authentication key, which is only accessible with a (free) account. In order to receive your own API key

- go to the OpenML website and log into your account
- then go to http://www.openml.org/u#!api.

For *demonstration purposes*, we have created a *public read-only API key* (`"c1994bdb7ecb3c6f3c8f3b35f4b47f1f"`), which will be used in the following to make the examples executable.

Permanently setting configuration

After registering, you can create a configuration file. The `config` file may contain the following information:

- `server`:
  - default: `http://www.openml.org/api/v1`
- `cachedir`:
  - directory where the current content of the cache is stored
  - the default cache directory can be obtained by the R command `file.path(tempdir(), "cache")`.
- `verbosity`:
  - `0`: normal output
  - `1`: info output (default)
  - `2`: debug output
- `arff.reader`:
  - `RWeka`: this is the standard Java parser used in Weka
  - `farff`: the farff package provides a newer, faster parser without any Java requirements
- `confirm.upload`:
  - default decision w.r.t. confirming uploads
  - per default (`FALSE`) one does not need to confirm the upload decision
- `apikey`:
  - required to access the server

The configuration file is *not mandatory*. Yet, permanently setting your API key via a `config` file is recommended, as this key is required to access the OpenML server. However, it is noteworthy that basically everybody who has access to your computer can read the configuration file and thus see your API key. With your API key other users have full access to your account via the API, so please handle it with care!

The configuration file and some related things are also explained in the OpenML Wiki.

Creating the configuration file in R

To set up your OpenML configuration, you can either use `setOMLConfig` or `saveOMLConfig`. The difference between those two commands is that `setOMLConfig` sets your configuration temporarily for the current R session, whereas `saveOMLConfig` saves the configuration permanently. In order to create a permanent configuration file using default values and at the same time setting your personal API key, run

```
saveOMLConfig(apikey = "c1994bdb7ecb3c6f3c8f3b35f4b47f1f")
```

where `"c1994bdb7ecb3c6f3c8f3b35f4b47f1f"` should be replaced with *your personal API key*.

Manually creating the configuration file

It is also possible to manually create a file `~/.openml/config` in your home directory – one can use the R command `path.expand("~/.openml/config")` to get the full path to the configuration file on the operating system. The `config` file consists of `key = value` pairs. An exemplary minimal `config` file might look as follows:

```
apikey=c1994bdb7ecb3c6f3c8f3b35f4b47f1f
```

Note that the values are not quoted.

If one manually modifies the `config` file, one needs to reload the modified `config` file to the current R session using `loadOMLConfig()`. You can query the current configuration using

```
getOMLConfig()
```

```
## OpenML configuration:
##   server           : http://test.openml.org/api/v1
##   cachedir         : L:\GitRepository\openml-r/tests/cache
##   verbosity        : 0
##   arff.reader      : RWeka
##   confirm.upload   : FALSE
##   apikey           : ****************************47f1f
```

As you can see, the configuration file lists the five items `server`, `cachedir`, `verbosity`, `arff.reader` and `apikey` that were listed in the beginning of this paragraph.

Once the config file is set up, you are **ready to go**!

Listing

In this stage, we want to list basic information about the various OpenML objects:

- data sets
- tasks
- flows
- runs
- run results
- evaluation measures
- task types

For each of these objects, we have a function to query the information, beginning with `listOML`. All of these functions return a `data.frame`, even in case the result consists of a single column or has zero observations (i.e., rows).

One should be aware of the fact that the `listOML*` functions only list information on the corresponding objects – they do not download the respective objects. Information on actually downloading specific objects is covered in the next section.

List data sets

To browse the OpenML data base for appropriate data sets, you can use `listOMLDataSets()` in order to get basic data characteristics (number of features, instances, classes, missing values, etc.) for each data set. By default, `listOMLDataSets()` returns only data sets that have an active status on OpenML:

```
datasets = listOMLDataSets()  # returns active data sets
```

The resulting `data.frame` contains the following information for each of the listed data sets:

- the data set ID `data.id`
- the `status` (`"active"`, `"in_preparation"` or `"deactivated"`) of the data set
- the `name` of the data set
- the size of the majority / biggest class (`majority.class.size`)
- etc.

```
str(datasets)
```

```
## 'data.frame':    2417 obs. of  17 variables:
##  $ data.id                              : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ name                                 : chr  "anneal" "anneal" "kr-vs-kp" "labor" ...
##  $ version                              : int  2 1 1 1 1 1 1 1 1 1 ...
##  $ status                               : chr  "active" "active" "active" "active" ...
##  $ format                               : chr  "ARFF" "ARFF" "ARFF" "ARFF" ...
##  $ tags                                 : chr  "1, capacapa, hallo, hallo1, joaquin, new_test, stu
##  $ majority.class.size                  : int  684 684 1669 37 245 813 57 NA 67 81 ...
##  $ max.nominal.att.distinct.values      : int  10 9 3 3 2 -1 6 -1 22 8 ...
##  $ minority.class.size                  : int  0 0 1527 20 0 734 1 NA 0 2 ...
##  $ num.binary.atts                      : int  14 7 34 3 73 0 61 0 4 9 ...
##  $ number.of.classes                    : int  6 6 2 2 16 26 24 NA 7 4 ...
##  $ number.of.features                   : int  39 39 37 17 280 17 70 7 26 19 ...
##  $ number.of.instances                  : int  898 898 3196 57 452 20000 226 345 205 148 ...
##  $ number.of.instances.with.missing.values: int  0 898 0 56 384 0 222 0 46 0 ...
##  $ number.of.missing.values             : int  0 22175 0 326 408 0 317 0 59 0 ...
##  $ number.of.numeric.features           : int  6 6 0 8 206 16 0 6 15 3 ...
##  $ number.of.symbolic.features          : int  32 32 36 8 73 0 69 1 10 15 ...
```

```
head(datasets[, 1:5])
```

```
##   data.id       name version status format
## 1       1     anneal       2 active   ARFF
## 2       2     anneal       1 active   ARFF
## 3       3   kr-vs-kp       1 active   ARFF
## 4       4      labor       1 active   ARFF
## 5       5 arrhythmia       1 active   ARFF
## 6       6     letter       1 active   ARFF
```

To find a specific data set, you can now query the resulting `datasets` object. Suppose we want to find the `iris` data set.

```
subset(datasets, name == "iris")
```

```
##     data.id name version status format                          tags
## 55       61 iris       1 active   ARFF study_1, study_4, study_7, uci
```

```
## 821     969 iris       3 active   ARFF               study_1, study_7
##     majority.class.size max.nominal.att.distinct.values
## 55                   50                              -1
## 821                 100                              -1
##     minority.class.size num.binary.atts number.of.classes
## 55                   50              0                 3
## 821                  50              0                 2
##     number.of.features number.of.instances
## 55                   5                150
## 821                  5                150
##     number.of.instances.with.missing.values number.of.missing.values
## 55                                        0                       0
## 821                                       0                       0
##     number.of.numeric.features number.of.symbolic.features
## 55                           4                           0
## 821                          4                           0
```

As you can see, there are two data sets called `iris`. We want to use the *original* data set with three classes, which is stored under the data set ID (`data.id`) 61. You can also have a closer look at the data set on the corresponding OpenML web page (http://openml.org/d/61).

List tasks

Each OpenML task is a bundle that encapsulates information on various objects:

- a specific type, e.g., `"Supervised Classification"` or `"Supervised Regression"`
- a data set
- a target feature (which might differ from the data set's default target)
- an estimation/resampling procedure, e.g., a 10-fold cross-validation
- data splits for this estimation procedure
- one or more (performance) evaluation measures, e.g., `"predictive accuracy"` for a classification task

Listing the tasks can be done via

```
tasks = listOMLTasks()
```

The resulting `data.frame` contains for each of the listed tasks information on:

- the task ID `task.id`
- the type of the task `task.type`
- information on the data set (analogously to the list data set area), such as the number of features, classes and instances
- the name of the target variable `target.feature`
- `tags` which can be used for labelling the task
- the `estimation.procedure` (aka resampling strategy)
- the `evaluation.measures` used for measuring the performance of the learner / flow on the task

```
str(tasks)
```

```
## 'data.frame':    5000 obs. of  26 variables:
##  $ task.id                          : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ task.type                        : chr  "Supervised Classification" "Supervised Classificati
##  $ data.id                          : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ name                             : chr  "anneal" "anneal" "kr-vs-kp" "labor" ...
##  $ status                           : chr  "active" "active" "active" "active" ...
##  $ format                           : chr  "ARFF" "ARFF" "ARFF" "ARFF" ...
##  $ estimation.procedure             : chr  "10-fold Crossvalidation" "10-fold Crossvalidation"
```

```
##  $ evaluation.measures                    : chr  "predictive_accuracy" "predictive_accuracy" "predict
##  $ target.feature                         : chr  "class" "class" "class" "class" ...
##  $ cost.matrix                            : chr  NA NA NA NA ...
##  $ source.data.labeled                    : chr  NA NA NA NA ...
##  $ target.feature.event                   : chr  NA NA NA NA ...
##  $ target.feature.left                    : chr  NA NA NA NA ...
##  $ target.feature.right                   : chr  NA NA NA NA ...
##  $ tags                                   : chr  "basic, study_1, study_7, under100k, under1m" "at2,
##  $ majority.class.size                    : int  684 684 1669 37 245 813 57 NA 67 81 ...
##  $ max.nominal.att.distinct.values        : int  10 9 3 3 2 -1 6 -1 22 8 ...
##  $ minority.class.size                    : int  0 0 1527 20 0 734 1 NA 0 2 ...
##  $ num.binary.atts                        : int  14 7 34 3 73 0 61 0 4 9 ...
##  $ number.of.classes                      : int  6 6 2 2 16 26 24 NA 7 4 ...
##  $ number.of.features                     : int  39 39 37 17 280 17 70 7 26 19 ...
##  $ number.of.instances                    : int  898 898 3196 57 452 20000 226 345 205 148 ...
##  $ number.of.instances.with.missing.values: int  0 898 0 56 384 0 222 0 46 0 ...
##  $ number.of.missing.values               : int  0 22175 0 326 408 0 317 0 59 0 ...
##  $ number.of.numeric.features             : int  6 6 0 8 206 16 0 6 15 3 ...
##  $ number.of.symbolic.features            : int  32 32 36 8 73 0 69 1 10 15 ...
```

For some data sets, there may be more than one task available on the OpenML server. For example, one can look for `"Supervised Classification"` tasks that are available for data set 61 via

```
head(subset(tasks, task.type == "Supervised Classification" & data.id == 61L)[, 1:5])
```

```
##      task.id                task.type data.id name status
## 53        59 Supervised Classification      61 iris active
## 271      289 Supervised Classification      61 iris active
## 442     1823 Supervised Classification      61 iris active
## 551     1939 Supervised Classification      61 iris active
## 598     1992 Supervised Classification      61 iris active
## 4445    7306 Supervised Classification      61 iris active
```

List flows

A flow is the definition and implementation of a specific algorithm workflow or script, i.e., a flow is essentially the code / implementation of the algorithm.

```
flows = listOMLFlows()
str(flows)
```

```
## 'data.frame':    4235 obs. of  7 variables:
##  $ flow.id         : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ full.name       : chr  "openml.evaluation.EuclideanDistance(1.0)" "openml.evaluation.PolynomialKe
##  $ name            : chr  "openml.evaluation.EuclideanDistance" "openml.evaluation.PolynomialKernel"
##  $ version         : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ external.version: chr  "" "" "" "" ...
##  $ uploader        : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ tags            : chr  "weka, weka_3.7.12" "" "" "" ...
```

```
flows[56:63, 1:4]
```

```
##    flow.id            full.name              name version
## 56      56         weka.ZeroR(1)         weka.ZeroR       1
## 57      57          weka.OneR(1)          weka.OneR       1
## 58      58    weka.NaiveBayes(1)    weka.NaiveBayes       1
```

```
## 59      59          weka.JRip(1)          weka.JRip       1
## 60      60           weka.J48(1)           weka.J48       1
## 61      61       weka.REPTree(1)       weka.REPTree       1
## 62      62 weka.DecisionStump(1) weka.DecisionStump       1
## 63      63 weka.HoeffdingTree(1) weka.HoeffdingTree       1
```

List runs and run results

A run is an experiment, which is executed on a given combination of task, flow and setup (i.e., the explicit parameter configuration of a flow). The corresponding results are stored as a run result. Both objects, i.e., runs and run results, can be listed via `listOMLRuns` or `listOMLRunEvaluations`, respectively. As each of those objects is defined with a task, setup and flow, you can extract runs and run results with specific combinations of `task.id`, `setup.id` and/or `flow.id`. For instance, listing all runs for task 59 (supervised classification on iris) can be done with

```
runs = listOMLRuns(task.id = 59L)  # must be specified with the task, setup and/or implementation ID
head(runs)
```

```
##   run.id task.id setup.id flow.id uploader error.message tags
## 1     81      59       12      67        1          <NA>
## 2    161      59       13      70        1          <NA>
## 3    234      59        1      56        1          <NA>
## 4    447      59        6      61        1          <NA>
## 5    473      59       18      77        1          <NA>
## 6    491      59        7      62        1          <NA>
```

```
# one of the IDs (here: task.id) must be supplied
run.results = listOMLRunEvaluations(task.id = 59L)
str(run.results)
```

```
## 'data.frame':    2511 obs. of  36 variables:
##  $ run.id                       : int  81 161 234 447 473 491 550 6088 6157 6158 ...
##  $ task.id                      : int  59 59 59 59 59 59 59 59 59 59 ...
##  $ setup.id                     : int  12 13 1 6 18 7 16 11 12 3 ...
##  $ flow.id                      : int  67 70 56 61 77 62 75 66 67 58 ...
##  $ flow.name                    : chr  "weka.BayesNet_K2(1)" "weka.SMO_PolyKernel(1)" "weka.ZeroR(1)"
##  $ data.name                    : chr  "iris" "iris" "iris" "iris" ...
##  $ upload.time                  : chr  "2014-04-07 00:05:11" "2014-04-07 00:55:32" "2014-04-07 01:33
##  $ area.under.roc.curve         : num  0.983 0.977 0.5 0.967 0.978 ...
##  $ average.cost                 : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ f.measure                    : num  0.94 0.96 0.167 0.927 0.947 ...
##  $ kappa                        : num  0.91 0.94 0 0.89 0.92 0.5 0.95 0.93 0.91 0.93 ...
##  $ kb.relative.information.score: num  1.39e+02 9.09e+01 -6.80e-05 1.31e+02 1.38e+02 ...
##  $ mean.absolute.error          : num  0.0384 0.2311 0.4444 0.0671 0.0392 ...
##  $ mean.prior.absolute.error    : num  0.444 0.444 0.444 0.444 0.444 ...
##  $ number.of.instances          : num  150 150 150 150 150 150 150 150 150 150 ...
##  $ precision                    : num  0.94 0.96 0.111 0.927 0.947 ...
##  $ predictive.accuracy          : num  0.94 0.96 0.333 0.927 0.947 ...
##  $ prior.entropy                : num  1.58 1.58 1.58 1.58 1.58 ...
##  $ recall                       : num  0.94 0.96 0.333 0.927 0.947 ...
##  $ relative.absolute.error      : num  0.0863 0.52 1 0.151 0.0881 ...
##  $ root.mean.prior.squared.error: num  0.471 0.471 0.471 0.471 0.471 ...
##  $ root.mean.squared.error      : num  0.16 0.288 0.471 0.211 0.178 ...
##  $ root.relative.squared.error  : num  0.339 0.611 1 0.447 0.377 ...
##  $ scimark.benchmark            : num  NA NA NA NA NA NA NA NA NA NA ...
```

```
##  $ total.cost                   : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ usercpu.time.millis          : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ usercpu.time.millis.testing  : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ usercpu.time.millis.training : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ area.under.roc.curve.array   : chr  "1,0.975,0.9744" "1,0.96,0.9704" "0.5,0.5,0.5" "1,0.9481,0.953
##  $ confusion.matrix.array       : chr  "50,0,0,0,46,4,0,5,45" "50,0,0,0,48,2,0,4,46" "50,0,0,50,0,0,5
##  $ f.measure.array              : chr  "1,0.910891,0.909091" "1,0.941176,0.938776" "0.5,0,0" "1,0.888
##  $ number.of.instances.array    : chr  "50,50,50" "50,50,50" "50,50,50" "50,50,50" ...
##  $ os.information.array         : chr  NA NA NA NA ...
##  $ precision.array              : chr  "1,0.901961,0.918367" "1,0.923077,0.958333" "0.333333,0,0" "1
##  $ recall.array                 : chr  "1,0.92,0.9" "1,0.96,0.92" "1,0,0" "1,0.88,0.9" ...
##  $ scimark.benchmark.array      : chr  NA NA NA NA ...
```

List evaluation measures and task types

Analogously to the previous listings, one can list further objects simply by calling the respective functions.

```
listOMLDataSetQualities()
listOMLEstimationProcedures()
listOMLEvaluationMeasures()
listOMLTaskTypes()
```

Downloading

Users can download data sets, tasks, flows and runs from the OpenML server. The package provides special representations for each object, which will be discussed here.

Download an OpenML data set

To directly download a data set, e.g., when you want to run a few preliminary experiments, one can use the function `getOMLDataSet`. The function accepts a data set ID as input and returns the corresponding `OMLDataSet`:

```
iris.data = getOMLDataSet(data.id = 61L)  # the iris data set has the data set ID 61
```

Download an OpenML task

The following call returns an OpenML task object for a supervised classification task on the iris data:

```
task = getOMLTask(task.id = 59L)
task
```

```
##
## OpenML Task 59 :: (Data ID = 61)
##   Task Type            : Supervised Classification
##   Data Set             : iris :: (Version = 1, OpenML ID = 61)
##   Target Feature(s)    : class
##   Tags                 : basic, study_1, study_7, under100k, under1m
##   Estimation Procedure : Stratified crossvalidation (1 x 10 folds)
##   Evaluation Measure(s): predictive_accuracy
```

The corresponding `"OMLDataSet"` object can be accessed by

```
task$input$data.set
```

```
##
## Data Set "iris" :: (Version = 1, OpenML ID = 61)
##   Collection Date         : 1936
##   Creator(s)              : R.A. Fisher
##   Default Target Attribute: class
```

and the class of the task can be shown with the next line

```
task$task.type
```

```
## [1] "Supervised Classification"
```

Also, it is possible to extract the data set itself via

```
iris.data = task$input$data.set$data
head(iris.data)
```

```
##   sepallength sepalwidth petallength petalwidth       class
## 0         5.1        3.5         1.4        0.2 Iris-setosa
## 1         4.9        3.0         1.4        0.2 Iris-setosa
## 2         4.7        3.2         1.3        0.2 Iris-setosa
## 3         4.6        3.1         1.5        0.2 Iris-setosa
## 4         5.0        3.6         1.4        0.2 Iris-setosa
## 5         5.4        3.9         1.7        0.4 Iris-setosa
```

Download an OpenML flow

Aside from tasks and data sets, one can also download flows – by calling `getOMLFlow` with the specific `flow.id`

```
flow = getOMLFlow(flow.id = 2700L)
flow
```

```
##
## Flow "classif.randomForest" :: (Version = 47, Flow ID = 2700)
##  External Version         : R_3.1.2-734b029d
##  Dependencies             : mlr_2.9, randomForest_4.6.12
##  Number of Flow Parameters: 16
##  Number of Flow Components: 0
```

Download an OpenML run

To download the results of one run, including all server and user computed metrics, you have to define the corresponding run ID. For all runs that are actually related to the task, the corresponding ID can be extracted from the `runs` object, which was created in the previous section. Here we use a run of task 59, which has the `run.id` 525534. Single OpenML runs can be downloaded with the function `getOMLRun`:

```
task.list = listOMLRuns(task.id = 59L)
task.list[281:285, ]
```

```
##     run.id task.id setup.id flow.id uploader error.message tags
## 281  14926      59      271     185       17          <NA>
## 282  14927      59      272     185       17          <NA>
## 283  14928      59      273     185       17          <NA>
## 284  14929      59      274     185       17          <NA>
```

```
## 285  14930      59      275     185       17          <NA>
```

```
run = getOMLRun(run.id = 524027L)
run
```

```
##
## OpenML Run 524027 :: (Task ID = 59, Flow ID = 2393)
##  User ID  : 970
##  Learner  : classif.randomForest(43)
##  Task type: Supervised Classification
```

Each `OMLRun` object is a list object, which stores additional information on the run. For instance, the flow of the previously downloaded run has some non-default settings for hyperparameters, which can be obtained by:

```
run$parameter.setting  # retrieve the list of parameter settings
```

```
## $seed
##  seed = 1
##
## $kind
##  kind = Mersenne-Twister
##
## $normal.kind
##  normal.kind = Inversion
```

If the underlying flow has hyperparameters that are different from the default values of the corresponding learner, they are also shown, otherwise the default hyperparameters are used (but not explicitly listed).

All the data that served as input for the run, including data set IDs and the URL to the data, is stored in `input.data`:

```
run$input.data
```

```
##
## ** Data Sets **
##   data.id name                                                        url
## 1      61 iris http://www.openml.org/data/download/61/dataset_61_iris.arff
##
## ** Files **
## data frame with 0 columns and 0 rows
##
## ** Evaluations **
## data frame with 0 columns and 0 rows
```

Predictions made by an uploaded run are stored within the `predictions` element and can be retrieved via

```
head(run$predictions, 10)
```

```
##    repeat fold row_id      prediction           truth
## 1       0    0     43     Iris-setosa     Iris-setosa
## 2       0    0     14     Iris-setosa     Iris-setosa
## 3       0    0     37     Iris-setosa     Iris-setosa
## 4       0    0     23     Iris-setosa     Iris-setosa
## 5       0    0     10     Iris-setosa     Iris-setosa
## 6       0    0     99 Iris-versicolor Iris-versicolor
## 7       0    0     87 Iris-versicolor Iris-versicolor
## 8       0    0     97 Iris-versicolor Iris-versicolor
## 9       0    0     62 Iris-versicolor Iris-versicolor
## 10      0    0     92 Iris-versicolor Iris-versicolor
```

```
##    confidence.Iris-setosa confidence.Iris-versicolor
## 1                       1                           0
## 2                       1                           0
## 3                       1                           0
## 4                       1                           0
## 5                       1                           0
## 6                       0                           1
## 7                       0                           1
## 8                       0                           1
## 9                       0                           1
## 10                      0                           1
##    confidence.Iris-virginica
## 1                          0
## 2                          0
## 3                          0
## 4                          0
## 5                          0
## 6                          0
## 7                          0
## 8                          0
## 9                          0
## 10                         0
```

The output above shows predictions, ground truth information about classes and task-specific information, e.g., about the confidence of a classifier (for every observation) or in which fold a data point has been placed.

Running

The modularized structure of OpenML allows to apply the implementation of an algorithm to a specific task and there exist multiple possibilities to do this.

Run a task with a specified mlr learner

If one is working with **mlr**, one can specify an `RLearner` object and use the function `runTaskMlr` to create the desired `"OMLMlrRun"` object. The `task` is created the same way as in the previous sections:

```
task = getOMLTask(task.id = 59L)

library("mlr")
lrn = makeLearner("classif.rpart")
run.mlr = runTaskMlr(task, lrn)
run.mlr
```

```
## $run
##
## OpenML Run NA :: (Task ID = 59, Flow ID = NA)
##
## $bmr
##   task.id    learner.id acc.test.join timetrain.test.sum
## 1    iris classif.rpart          0.94               0.07
##   timepredict.test.sum
## 1                    0
##
## $flow
```

```
##
## Flow "mlr.classif.rpart" :: (Version = NA, Flow ID = NA)
##  External Version         : R_3.3.2-v2.2b256c3
##  Dependencies             : R_3.3.2, OpenML_1.0, mlr_2.10, rpart_4.1.10
##  Number of Flow Parameters: 13
##  Number of Flow Components: 0
##
## attr(,"class")
## [1] "OMLMlrRun"
```

Note that locally created runs don't have a run ID or flow ID yet. These are assigned by the OpenML server after uploading the run.

Run a task without using mlr

If you are not using `mlr`, you will have to invest some more time and effort to get things done since this is not supported yet. So, unless you have good reasons to do otherwise, we strongly encourage to use `mlr`. If the algorithm you want to use is not integrated in `mlr` yet, you can integrate it yourself (see the tutorial) or open an issue on mlr GitHub repository and hope someone else will do it for you.

Uploading

The following section gives an overview on how one can contribute building blocks (i.e. data sets, flows and runs) to the OpenML server.

Upload a data set

A data set contains information that can be stored on OpenML and used by OpenML tasks and runs. In this example, a very simple data set is taken from R, converted to an OpenML data set and afterwards uploaded to the server. The corresponding workflow of uploading a data set consists of the following three steps:

1. `makeOMLDataSetDescription`: create the description object of an OpenML data set
2. `makeOMLDataSet`: convert the data set into an OpenML data set
3. `uploadOMLDataSet`: upload the data set to the server

```
## This is simple data set, which is generally already available in R.
## Please DO NOT upload it to the server!
data("airquality")
dsc = "Daily air quality measurements in New York, May to September 1973.
  This data is taken from R."
cit = "Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983)
  Graphical Methods for Data Analysis. Belmont, CA: Wadsworth."

## (1) Create the description object
desc = makeOMLDataSetDescription(name = "airquality",
  description = dsc,
  creator = "New York State Department of Conservation (ozone data) and the National
    Weather Service (meteorological data)",
  collection.date = "May 1, 1973 to September 30, 1973",
  language = "English",
  licence = "GPL-2",
  url = "https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/00Index.html",
  default.target.attribute = "Ozone",
```

```
  citation = cit,
  tags = "R")

## (2) Create the OpenML data set
air.data = makeOMLDataSet(desc = desc,
  data = airquality,
  colnames.old = colnames(airquality),
  colnames.new = colnames(airquality),
  target.features = "Ozone")

## (3) Upload the OpenML data set to the server
dataset.id = uploadOMLDataSet(air.data)
dataset.id
```

Alternatively you can enter data directly on the OpenML website.

Upload a flow

A flow is an implementation of a single algorithm or a script. Each `mlr` learner can be considered an implementation of a flow, which can be uploaded to the server with the function `uploadOMLFlow`. If the flow has already been uploaded to the server (either by you or someone else), one receives a message that the flow already exists and the `flow.id` is returned from the function. Otherwise, the flow will be uploaded, receive its own `flow.id` and return that ID.

```
library("mlr")
lrn = makeLearner("classif.randomForest")
flow.id = uploadOMLFlow(lrn)
flow.id
```

Upload a run

In addition to uploading data sets or flows, one can also upload runs (which a priori have to be created, e.g., using `mlr`):

```
## choose 2 flows (i.e., mlr-learners)
learners = list(
  makeLearner("classif.kknn"),
  makeLearner("classif.randomForest")
)

## pick 3 random tasks
task.ids = c(57, 59, 2382)

for (lrn in learners) {
  for (id in task.ids) {
    task = getOMLTask(id)
    res = runTaskMlr(task, lrn)$run
    run.id = uploadOMLRun(res)  # upload results
  }
}
```

Before your run will be uploaded to the server, `uploadOMLRun` checks whether the flow that created this run is already available on the server. If the flow does not exist on the server, it will (automatically) be uploaded as well.

Feedback

Now, you should have gotten an idea on how to use our package. However, as there is always room for improvement, we are more than happy to receive your feedback. So, in case

- there is anything not well documented
- you encounter a bug
- are missing functionality

please open an issue in the issue tracker of our GitHub repository.