Universiteit Leiden
The Netherlands

Opleiding Informatica

OpenML-Connect: A C++ connection library for OpenML

Thomas Wink

Supervisors:
Dr. J.N. van Rijn & Dr. W.A. Kosters

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

08/08/2020

Abstract

Reproducible results are one of the cornerstones of open science. In an environment of large datasets and comprehensive data analysis, experimental results are quickly getting more difficult to reproduce. OpenML aims to solve this problem, by providing a place to not only store datasets, but also all machine learning algorithms that have been used on these datasets and the results that these algorithms produced.

OpenML has previously been connected with both machine learning tools and packages through its REST API. In this thesis, we present a C++ library that contains a connector to easily connect to OpenML's API and access the data it provides. We performed a case study involving an algorithm selection problem to demonstrate its use. In this study, we found that the trained model performed better than two alternative baseline methods that we used. This adds to the empirical proofs that suggest the benefits of algorithm selection.

Throughout these experiments, and during the testing of the library, the connector performed its tasks well, although there still is quite some functionality that can be implemented.

Contents

1 Introduction

In many scientific communities, large datasets are important or their importance is rising. Various operations are performed on them, all kinds of machine learning algorithms can be used to categorize and classify this information and various conclusions are substantiated by these datasets. Then a paper is written about these findings, but in times where the communication of science is becoming more important every day, the results and the conclusions are presented very well, whereas the analysis of the datasets is sometimes ignored or underrepresented. This threatens one of the pillars of modern science, the verifiability and reproducability of research.
To address this problem, OpenML.org [VvRBT13] was started. This web platform can be used to not only store the databases used for research, but also to store the experiments that have been run on these datasets and the results of these experiments. As can be seen in Figure 1, this follows a specific pattern. We will explain OpenML in further detail in Section 3.

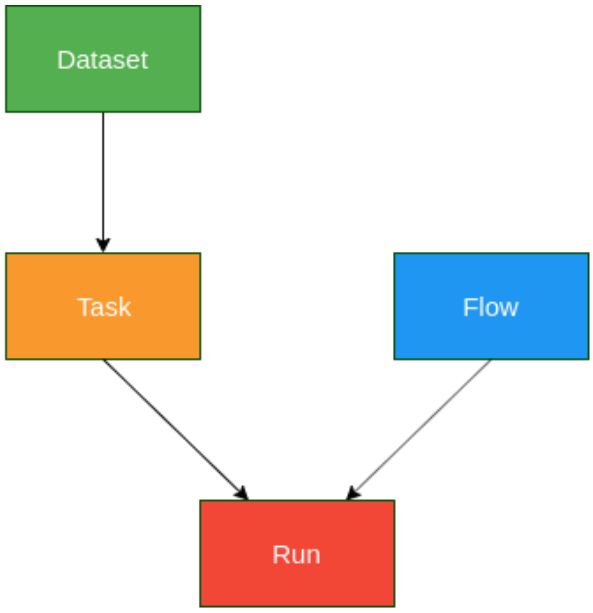

Figure 1: Workflow that is used in OpenML. (Source: https://towardsdatascience.com/openml-machine-learning-as-a-community-d678306e1a7e)

With not only the results of the experiments available to everyone, but also the datasets, the studies using this platform have taken a large step towards becoming more reproducible.

Another benefit of OpenML is its integration in several machine learning environments, like Weka [WFHP16]. This means that the environment will automatically download the data from OpenML, run the given algorithm and upload the results to the OpenML servers. There are some Application Programming Interfaces (APIs) available for several other environments and languages, which make it easy to download datasets, all kinds of metadata and results of previous runs. These APIs are built using the OpenML REST API, which makes it possible to communicate directly with the OpenML servers. These integrations and APIs ensure a good user-experience for a large number of possible users. We will expand on these APIs in Section 3.3. Our goal is to make a library for C++

to connect to OpenML's REST API and test this library through a case study.

1.1 Contributions

In this thesis, we present a library for C++ that enables its users easy access to some of the key features available in the OpenML API. This library, also called a *connector*, will:

- take care of the connection with the OpenML servers;
- parse downloaded information;
- store and present information in C++ objects generally using the same structure as in the OpenML workflow.

The source code of this library is publicly available on Github[1]. We also provide a small machine learning case study as a proof of concept. This case study will consist of an algorithm selection problem.

1.2 Thesis overview

In Section 2, we will discuss previous work on this subject. In Section 3, we will give some background information regarding machine learning and OpenML. In Section 4, we will discuss the implementation of the connector. In Section 5, we will explain the performed experiments and look at the results. In Section 6, we will discuss the findings of this project and evaluate the connector. Here, we will also take a look into the future and see what further research is possible on this subject. This research was carried out as a Bachelor project for the Computer Science program at Leiden Institute of Advanced Computer Science (LIACS), Leiden University. This thesis was written under the supervision of dr. J.N. van Rijn and dr. W.A. Kosters.

[1] https://github.com/ThomasWink/OpenML-Connect

2 Related Work

As all research, this thesis is built on previous work on the subject. Machine learning is not a new concept and much research has already been done. Consequently, this is a summary of the work this thesis is based on.

In [GC08], we have quite some background information on metalearning, as well as an outline of how metalearning systems work. We can use the frameworks described here as a basis for our own system. Although this is an older paper, the concepts presented and summarised here are still largely applicable.

Metalearning is based on the characterizations, or metafeatures, of datasets. These metafeatures are predictive for the performance of machine learning algorithms. In [RGS+18], the authors propose a systematisation and standardisation for data characterisation. A tool for extracting metafeatures from datasets is also proposed. This tool, the Meta-Feature Extractor (MFE), is explained in [ASR+20]. Currently, only classification datasets can be characterised by MFE.

At the same time, in [JSG19] a model is described where tabular datasets are visualised is a hierarchical set representation. In this way it is possible to extract dataset metafeatures by enforcing a proximity in the representation for similar datasets. This model, Dataset2Vec, is able to generalize beyond the different schemata of datasets.

Previously, the authors of [CBL+17] made a connector for R. This connector uses mlr [BLK+16] as a supporting package and allows mlr learners to be easily used on OpenML tasks.

In [FvK+19], the authors created a connector for Python. In Section 4.2, we will compare the functionality implemented in our connector to this connector. This connector also has a plugin interface that standardises interaction between machine learning library code and OpenML-Python. Subsequently they made a plugin to interface to scikit-learn, as that is one of the most popular machine learning libraries in Python.

3 Background

In this section, we will explain the basics of machine learning and how these basics tie in with the workflow of OpenML. We will also look into the differences between machine learning on different platforms. An important thing to note is the difference between a machine learning algorithm and a model. The difference is that a machine learning algorithm receives data and trains a model, which can then be used to perform predictions for unseen examples.

3.1 Machine Learning

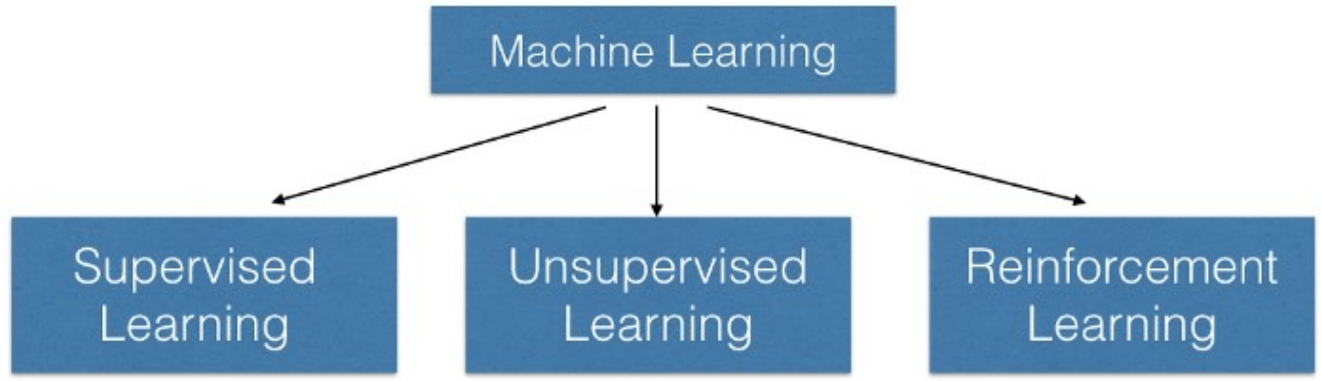

Figure 2: The three approaches to machine learning. (Source: https://miro.medium.com/max/1165/1*5yz2QF9_EHzttwa7Iyd2zw.png)

As mentioned previously, OpenML is an online platform where machine learning datasets, experiments and results can be posted, stored and used. OpenML follows a certain workflow. This workflow is based on the way machine learning works. As can be seen in Figure 2, there are three approaches to machine learning. In reinforcement learning, software agents are learning how to behave and react to a (changing) environment. This is achieved by giving this agent either a positive or negative reward upon an action it has performed [NR09]. In unsupervised learning, an algorithm gets an unlabeled dataset. This dataset is then either used to train a certain type of neural network, it is classified using clustering, or dimensionality reduction is performed [Bis06]. However, as both reinforcement learning and unsupervised learning are not used in this thesis, we will not elaborate on them.

The third approach to machine learning is supervised learning. Just as with unsupervised learning, these algorithms receive a dataset, but this dataset is labeled. This means that for all instances in the dataset it is known what the output is supposed to be. Supervised learning includes both regression and classification algorithms. Regression algorithms create a model that predicts some

form of numerical value for each instance in the dataset, whereas in classification algorithms the created model predicts a class to which an instance belongs. In this thesis, we will focus on classification algorithms, as we are focused on a classification based algorithm selection problem. However, the connector is by no means designed to be limited to these algorithms.

The model which is created by a classification algorithm first has to be trained. This is done during the training phase, when it receives some data to train its model on. Each algorithm has a set of parameters that can be tuned to improve the performance of the model it creates on this dataset. Optimal parameters can be different for each dataset.

In the category of classification algorithms there is a large variety of algorithms that can be used. Every algorithm performs differently, and can have some additional advantages and disadvantages. Some algorithms need a large dataset, where other algorithms need quite some time to train their model. Every algorithm has its own expertise, and performs well on different types of datasets. This means that always using the same algorithm is not profitable performance-wise. This is where algorithm selection problems come from.

3.2 OpenML workflow

The workflow in OpenML is split into four parts, as can be seen in Figure 1. It all starts with the dataset. This dataset does not stand on its own, but some metadata about the dataset is also provided. Every user can upload datasets, and if a dataset is made public everyone can see it. Datasets can have *qualities*, which is the name OpenML uses for metafeatures.

The next step is to define a *task*. There can be many tasks for each dataset, as a task is the goal to be reached for a dataset. This can for instance be "Supervised Classification on iris". Every user can define tasks on all public datasets, and every user can see these tasks.

The *flows* in OpenML are the algorithms that are available to the users. The names of flows are a combination of the package they are in and the name of the algorithm, as a Random Forest in Weka could perform differently compared to a Random Forest in mlr. A flow is purely an algorithm, and none of the parameters are defined.

The last component of the OpenML workflow is the *run*. Runs are flows with a set of parameters, applied to a task. On the task-view, as can be seen in Figure 3, there is an overview of the top 100 best flows for that task. This top 100 is based on a metric that can be chosen by the user. Then each flow's rank is chosen by the best performing run.

3.3 Machine Learning on different platforms

As mentioned in Section 1, there are connectors from languages and from machine learning applications to OpenML. These are called connectors and integrations respectively. Connectors provide functionality to make use of OpenML through its API. These functions are provided through

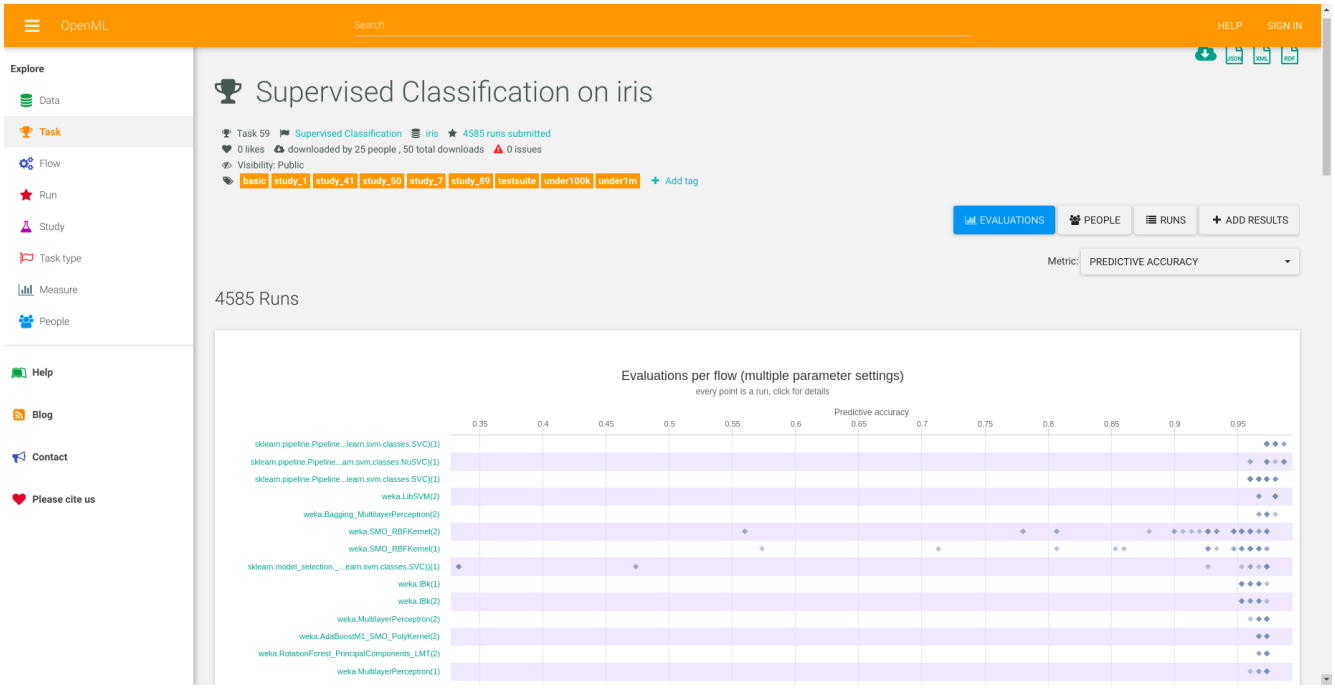

Figure 3: A task on OpenML. Here we see the performance of the best runs that were executed on this task. (Source: https://www.openml.org/t/59)

packages or libraries. Integrations are the instances where OpenML is interwoven in the application. In this way, you can, for instance, automatically upload your results to OpenML. Both connectors and integrations make use of OpenMLs REST API.

OpenML does not contain any algorithms. The algorithms that are referenced in OpenML are from machine learning packages and programs. Through the aforementioned connectors and the algorithms in machine learning packages such as scikit-learn [PVG+11] (Python) and mlr [BLK+16] (R) it is possible to highly automate the process of machine learning.

In C++, there are two main machine learning libraries. These are SHARK [IHMG08] and mlpack [CEL+18]. We had some problems with the integration of these as packages, as can be read in Section 4.4.1, so neither of these were actually used.

4 Implementation

In this section we describe how the connector is implemented. This does not only contain the funtioning of the connector itself, but also problems that were found along the way.

4.1 Components of the connector

The connector consists of four C++ classes to provide some services and a whole list of classes that are used to store the downloaded information. The four classes that provide these services are HttpConnection, XMLParser, Tagged and File. In Figure 4, the relationships between the first two classes is explained.

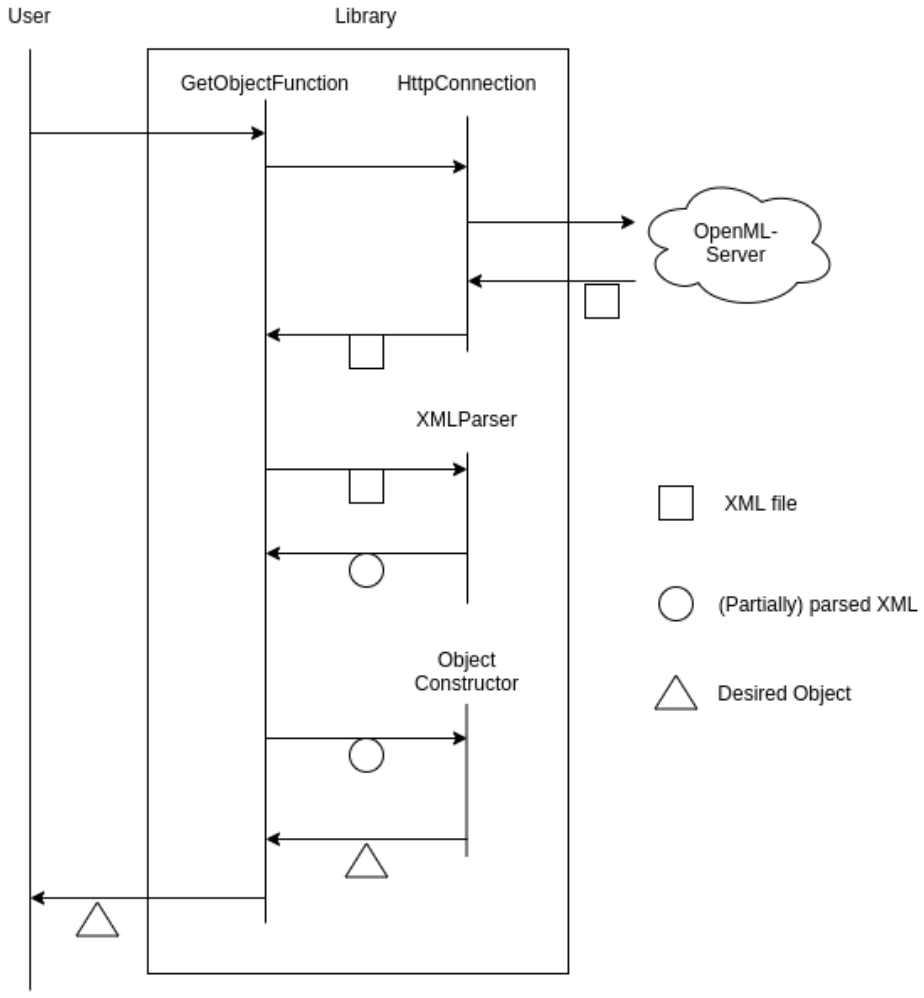

Figure 4: Sequence diagram showing the internal relations in the library and the interaction between user, library and OpenML servers. This diagram is generalized using a fictional function getObjectFunction, which is representative for 90% of implemented functionality.

In the class HttpConnection we have the functionality to upload dataset descriptions and to download all needed files from the OpenML servers. These downloaded files are then saved as a temporary file in the same directory as the executable, where it can be read by other components of the connector. This class relies heavily on the curl library [Ste18].

The class XMLParser contains functions to parse XML files from either a file or a given string. The main parsing function in this class constructs an unordered map that maps from a string (the XML tag) to a list of strings (the value of each of the instances of this tag). As each of these strings can be in XML, this is a way to recursively parse and store the given XML file. This class relies heavily on the RapidXml library [Kal06].

Tagged consists of a function that deletes all surplus XML tags around a given string. This function is used in so many information-storing classes that making a class specifically for this function was more space efficient.

The class File creates a file that is suitable for upload to OpenML as a dataset description. This dataset description is the most basic description possible and does not contain much information, but it does comply with the constraints of both XML and the requested information.

4.2 Implemented functionality

OpenML is an extensive website with a wide range of possible API calls. These calls are divided into several groups: data, task, task type, flow, setup, run, evaluation and study. There are some other small groups of calls that are mostly concerned with data about OpenML itself. We will not expand on these smaller groups. At this moment, a task type is a template of a task. Tasks were mostly concerned with classification problems, but are now considered legacy functions. As such, we use task inputs instead of tasks. Because we are working with classification problems, and task types are used for all other types of problems, task types are not in the scope of this project.

We focused on the design of a solid framework for the library. We did this because the complete list of functionality of the OpenML REST API is to large to fit completely in the scope of this project. Here, we list the functions as they are implemented in the library, and we compare them to functions in the OpenML Python API. Any parameters that are supposed to be given to the functions are omitted for the sake of clarity.
As can be seen in Table 1, the functions implemented in this library are mostly concerned with downloading information from the OpenML servers. This was due to several reasons. At first this started as these functions were the easier API-calls to perform. Gradually it became apparent that time constraints did not allow for the extra time needed to implement much of the functionality. This meant that we had to choose what to implement. We decided on this list of functions as they provide the most options to users with the smallest amount of implementation.

There are many API-calls yet to be implemented in this library. These calls can be placed in one of several categories. The calls regarding setups and studies are not implemented, as they were not essential with regards to a proof of concept experiment. Throughout all groups of calls named

C++ function name	Python function name	Description
uploadDataset()	datasets.create_dataset()	Creates metainformation from a given dataset and parameters and uploads the dataset and the metainformation to OpenML servers
getRunList()	runs.list_runs()	Returns a list of runs matching the given filters
getEvaluationList()	evaluations.list_evaluations()	Returns a list of evaluations matching the given filters
getTaskList()	task.list_tasks()	Returns a list of tasks matching the given filters
getTask()	task.get_task()	Returns a single task given the id of this task
getFlowList()	flow.list_flows()	Returns a list of flows matching the given filters
getFlow()	flow.get_flow()	Returns a single flow given the id of this flow
getDatasetList()	dataset.list_datasets()	Returns a list of datasets matching the given filters
getQualitiesList()	datasets.get_dataset()	Returns a list of qualities given the id of the dataset the qualities belong to. In Python this is included in the metainformation of the dataset
getDatasetDescription()	datasets.get_dataset()	Returns the metainformation given a dataset id. The Python version can also download the data itself

Table 1: List of implemented functions in openML-C++, with a comparable function from openML-Python and a description of the functionality.

at the start of this section, we did not implement deletion, tagging and untagging calls. All but one of the upload calls throughout the API are not implemented as they were not needed for the experiments. We did implement the call to upload a dataset to build, test and demonstrate the framework to perform such a task. Spread over all groups are some status-checking, status-updating or ownership-checking calls that were not implemented as they were outside the scope of this thesis.

4.3 Connector example

As an example, we will show one of the functions from the library and walk through all the actions that are performed to execute this function. We will take a deeper look into the function `getDatasetList()`. The code that is shown is condensed and is only an indication of the structure.

```
//OpenML.h
DatasetList getDatasetList (...){
    HttpConnection h;
    h.downloadToFile(...);
    XMLParser x;
    return DatasetList(x.parseFromFile(...));
}
```

Figure 5: Pseudocode of the function getDatasetList().

The code in Figure 5 is a shortened version of the information shown in Figure 4. We have already described what the first three lines of this function do, but we have not yet demonstrated how the information is then stored in their respective objects.

```
//DatasetList.h
class DatasetList {
    List<Dataset> dl;

    DatasetList (inputset is){
        XMLParser x;
        for(var i in is)
            Dataset d = Dataset(x.parseFromString(i));
            dl.push_back(d);
    }
}
```

Figure 6: Pseudocode of the class DatasetList.

In Figure 6, the building of a list of datasets is presented. These datasets are then constructed as described in Figure 7.

```
//Dataset.h
class Dataset {
    ...
    QualitiesList ql;

    Dataset (...){
        storeLocalInformation();
        ql = QualitiesList(...);
    }
}
```

Figure 7: Pseudocode of the class Dataset.

As can be seen in Figure 7, the `Dataset` objects are not datasets themselves, but rather they contain information about the dataset, such as the name, the format, its qualities and the dataset-id. This dataset-id can then be used to download the dataset description, which contains the URL where the dataset can be downloaded.

The `QualitiesList` that is present in a `Dataset` is constructed similarly to the `DatasetList`, but a `Quality` simply contains a name and a value, and as such does not contain a new layer of objects to be constructed.

4.4 Setbacks

During the project, we encountered a multitude of problems, both small and large. There were also some recurring problems that we had to deal with. This is a selection of problems we had to face.

4.4.1 Issues with the package management system

To manage the package management for the connector, we wanted to use a package management system. We decided on using Conan [con15]. Conan is an open-source decentralized package-manager. This system has a client-server architecture. The packages can be fetched from the server, and users can also create their own remote servers or add packages to other servers. We used this system to avoid problems with missing dependencies. However, when we started doing experiments, we found that one of the dependencies of the used machine learning package was not automatically included. When we manually imported this package, we found that one of the dependencies of the previous dependency was now not included, but this one could not be imported through Conan. We could have solved this by making our own remote Conan-server, but we instead decided to use the command line version of the machine learning package for the parts that needed this missing package. An integration with machine learning packages in C++ would be a good project to expand on this library.

4.4.2 Circular dependencies

Flows in OpenML can contain components, which can contain multiple flows. This creates a recursive dependency in the system, as the flow class can not be declared after the component class, because it can be a part of a component. This is solved by creating an abstract base class OMLPrimitive. Both components and flows inherit from this class and expand on its methods.

```
//Flow.h
#include "Component.h"
Flow::Flow(...){
    ...
}
...
```

Figure 8: Pseudocode of the Flow class.

In Figure 8, we first include `Component.h`. Then we elaborate on the class `Flow`.

```
//Component.h
#include "OMLPrimitive.h"
class Flow : public OMLPrimitive {
    ...
}
class Component : public OMLPrimitive {
    ...
}
```

Figure 9: Pseudocode of the Component class.

In Figure 9, we start by including `OMLPrimitive.h`. Then, we declare the class `Flow`, but we do not work out any of its functions. The class `Component` is worked out after the declaration of the functions of `Flow`.

```
//OMLPrimitive.h
class Flow;
class OMLPrimitive {
    ...
}
```

Figure 10: Pseudocode of the OMLPrimitive class.

In Figure 10, we start with a forward declaration of the class `Flow`. Then, we declare the virtual

```
<oml:flow xmlns:oml="http://openml.org/openml">
  <oml:id>300</oml:id>
  ...
  <oml:component>
    <oml:identifier>W</oml:identifier>
    <oml:flow xmlns:oml="http://openml.org/openml">
      <oml:id>58</oml:id>
      ...
    </oml:flow>
  </oml:component>
  ...
</oml:flow>
```

Figure 11: Example of an XML that contains recursive elements

class `OMLPrimitive`.

The way this is set up means that `OMLPrimitive` can use a pointer to a `Flow` as the return value of a function `recast`. This function is expanded on in `Flow.h`, where it recasts itself from a `OMLPrimitive` to a `Flow`. After this recast it also calls its own parse function. This means that when, in `Component.h`, a `Flow` must be made, it is first created as a pointer to a `Flow` stored in a pointer to an `OMLPrimitive`. Then it can be recast by calling that function on itself, and only then can it be stored as a pointer to a `Flow`.

4.4.3 Connection with OpenML

The connection with OpenML goes through their REST API. Although this API has a large documentation, some of the information is either obsolete or overly complicated. For instance, the OpenML task is considered as a legacy function. However, it can still be downloaded from the API and it has a large and cumbersome XML scheme. We solved this problem by using the Task Input, another XML-structure that was available in the API. However, this way of getting a Task Input is not documented. This situation can be avoided by maintaining the otherwise excellent API documentation.

4.4.4 Recursive parsing and storing

The XML files that are downloaded from the OpenML servers can be large and can theoretically contain an unbounded number of layers. This is because of the recursion that is present in some elements of the OpenML workflow, mainly in the flow. An example of this is shown in Figure 11. This would result in an enormous parser, which would have to recognise every special section by itself. To prevent this, we parse as little of the XML file as possible each time the parser is used. This is done by storing the XML tag together with the value of the tag using unordered maps from a string to list of strings. As a new layer is contained in a tag, this

part of the XML file can be stored as the value of the opening tag. Now, many of the special cases that are present in these XML files can be recognised in the function where the parser is called.

5 Case Study

In this section, we will take a look at the case study we have done. The goal of this case study is to evaluate the connector by doing a small metalearning experiment and using some of the functionality the connector offers.

5.1 Used data

To perform an experiment, one first needs data. OpenML, and thus also the connector, is a large source of metadata. The dataset used in the case study is related to a benchmark-suite that was presented in [BCF+17]. This benchmark-suite OpenML-CC18 contains a list of datasets and a small amount of information about them. However, as it contains the task ID, all information that is present in OpenML about these datasets and algorithms is available to us. This benchmark contains 72 curated and carefully selected datasets, but we use only 60 of those datasets. These 60 datasets had runs for each of the 10 flows we are comparing, but the remainder of the datasets missed runs for one or two of the flows. The 10 algorithms that are compared are shown in Figure 12.

In Table 2 we have a small example from our dataset. As can be seen, the Tasks are repeated, while several algorithms are executed on every task. This dataset contains 60 different tasks, and every task is performed with 10 different algorithms. The Task ID can be used to get the meta features that are used for the prediction in Table 3.

Task ID	Setup ID	Function	Flowname	Value
3	8255195	predictive_accuracy	AdaBoost Classifier	0.995932
6	8255228	predictive_accuracy	AdaBoost Classifier	0.88145
3	8254870	predictive_accuracy	Bernoulli NB	0.875156
6	8254876	predictive_accuracy	Bernoulli NB	0.4215
⋮	⋮	⋮	⋮	⋮

Table 2: Example from the case study dataset.

5.2 Tools and Experiment

The experiment we performed was a small metalearning experiment. In this experiment, we aimed to predict what type of algorithm can best be used to perform several classification problems. As this is also a classification problem, we can handle this as a supervised classification problem by finding the best performing algorithm for each dataset based on the predictive accuracy. Each row in the dataset we used consists of values for its qualities.

These qualities will be what the prediction will be based on. However, not every flow returns the same qualities. This means that we have to ensure that the qualities we feed to our model are comparable. We achieved this by searching for the qualities that are present in each run. As we

AdaBoost (AB) A meta-estimator that uses several weak learners to boost its own performance in an adaptive way. Additional copies of these "helpers" are used on the previously misqualified instances. [FS97]

Bernoulli NB (BNB) A Naive Bayes classifier geared towards Bernoulli or Boolean data. A Naive Bayes classifier is a classifier that uses Bayes' theorem with the assumption that every pair of features is independent. [MN98]

Decision Tree (DT) A classifier that generates a model with simple if-else rules. This model can easily be visualized as a decision tree. In these trees the leaves represent the predicted labels and the nodes represent the decisions to be made.[RM05]

Random Forest (RF) A classifier that takes a number of trees from a sample where each tree is made with a degree of randomness. This randomness is introduced to prevent the overfitting that decision trees are prone to.[Bre01]

Extra Trees (ET) Extra Trees is a Random Forest with some additional randomness. This is achieved by choosing the best of a set of randomly generated splitting rules.[GEW06]

Gradient Boosting (GB) Just like AdaBoost, Gradient Boosting is a boosting algorithm, which uses weak learners. In this case, these learners are selected from a sample based on the optimization of a loss function of a gradient in the function space.[Fri01]

k Neighbors (kN) An instance-based learning algorithm, which stores instances from the training set rather than making a model. When a new instance has to be classified it looks at its k nearest neighbors and votes on which class to belong to.[CH67]

Multi-layer Perceptron (MLP) A Neural Network with one or more hidden layers. This Multi-Layer Perceptron uses backpropagation to train.[RHW86]

Stochastic Gradient Descent (SGD) Not a classifier per se, but actually an optimisation technique. Stochastic Gradient Descent is a popular algorithm for training of both linear and non-linear models.[Bot98]

Support Vector Machine (SVM/SVC) When used as a classifier, a Support Vector Machine maps instances to a point in space where instances of different classes are linearly separated by as large a gap as possible. A Support Vector Machine support non-linear classification when it uses the so-called kernel trick. When used as a classifier the Support Vector Machine can also be called a Support Vector Classifier.[CV95]

Figure 12: List of algorithms that are compared in these experiments. These are the scikit-learn [PVG+11] implementations of these algorithms

wanted to handle this as a supervised classification problem, we needed a label to train the model with. The obvious candidate is the name of the flow that was used in the run. This results in a dataset that looks like the example in Table 3. To make this dataset, we have used several of the functions from the C++ connector. Both `getEvaluationList` and `getTask` were used to find the best flows for each dataset. Futhermore, `getQualitiesList` was used to gather all used qualities for each run.

Quality 1	Quality 2	Quality 3	...	Best flow
0.999061	0.99857	0.01157	...	Decision Tree
0.040902	4.69981	0.00085	...	Multilayer Perceptron
⋮	⋮	⋮	⋱	⋮

Table 3: Example from the training dataset

As was already mentioned in Section 4.4.1, there were some problems with dependencies, which resulted in the decision to use neither Shark or the C++ version of mlpack, but the Command Line Interface (CLI) of mlpack [CEL+18]. In this interface, it is possible to train models with a wide variety of settings.

The mlpack CLI has many possible commands to execute all kinds of algorithms. We will focus on the Random Forest algorithms, as that is what we used to make predictions. We decided on using a Random Forest because of its high accuracy [Bre01]. Normally, the training of the algorithm starts with splitting the dataset into a trainingset and a testset. However, this so-called preprocess splitter did not perform well on our dataset with qualities and changed every number into a zero. This was solved by quickly manually splitting the file in a trainingset and a testset.

The second step of the experiment is the training of the model. This is achieved with the following command:

```
mlpack_random_forest                    \
    --training_file train.csv           \
    --labels_file train.labels.csv      \
    --num_trees 100                     \
    --minimum_gain_split 0.1            \
    --minimum_leaf_size 3               \
    --print_training_accuracy           \
    --output_model_file rf-model.bin    \
    --verbose
```

This is the long version of the command, and every parameter has a shorthand.
The last step is to run the model on the test data and see how it performs. This can be done with the following command. As before, there is a shorthand for every parameter.

```
mlpack_random_forest                                    \
    --input_model_file rf-model.bin                     \
    --test_file test.csv                                \
    --test_labels_file test.labels.csv                  \
    --predictions_file predictions.csv                  \
    --verbose
```

5.3 Results

After all steps we took in Section 5.2, we had a list of predictions and a list of actual classes. When visualised in a confusion matrix this should, in the perfect world, become a diagonal line from the top left to the bottom right of the matrix. This would mean that every prediction is correct.

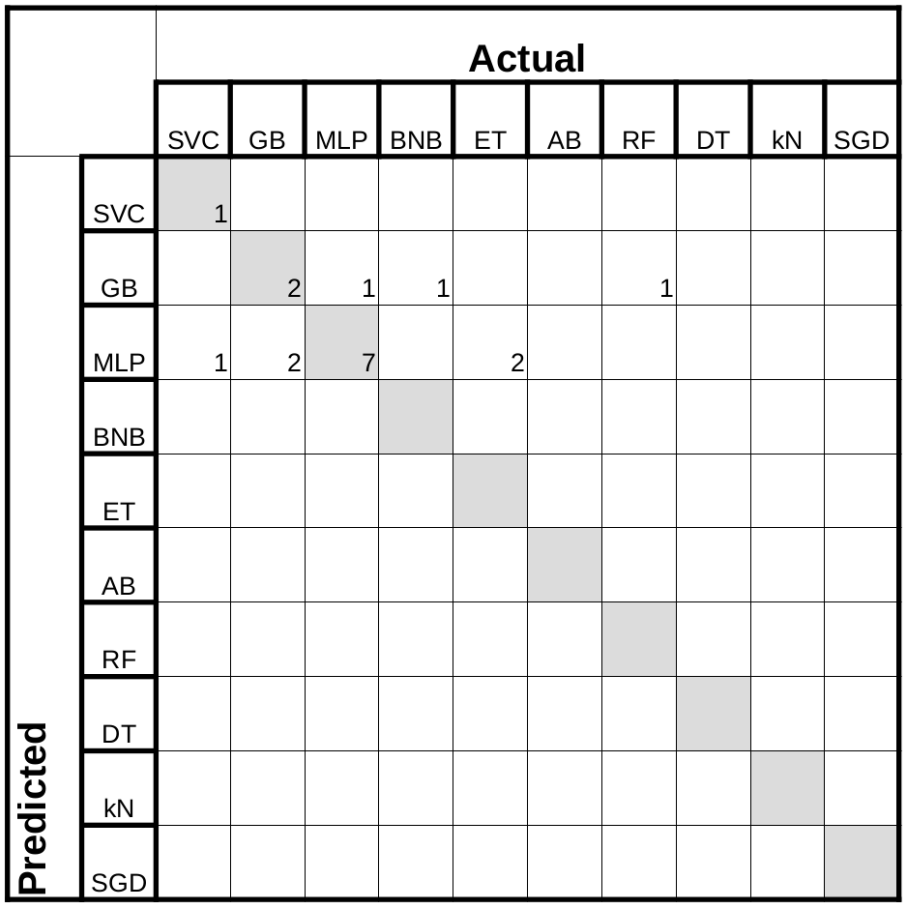

Figure 13: Confusion matrix made by a Random Forest model on which classifier to use on a dataset.

However, when we look at Figure 13 we see that this diagonal is not nearly as visible as it should be. Out of the 18 predictions made by our trained model, only 10 were correct. This is an accuracy

of 55% for these predictions. There are some interesting things to note about these results.

One of the things that is remarkable is that the Multi-Layer Perceptron is predicted 12 times, more than double that of the next most predicted classifier. This can be explained by the data itself, as the Multi-Layer Perceptron is the best classifier overal in these datasets. Another interesting thing we see here is the fact that our model only predicts 3 of the 10 algorithms. We have a suspicion that this is caused by the implementation of the Random Forest algorithm combined with our small dataset. The algorithm has not seen four of the available options as these classifiers were not present in the trainingset, and thus would not know they exist. We have not been able to find proof for this.

These results led to us further looking into the performance of our model. How does this trained model compare to some baseline methods? We decided on comparing our model with a random selection method and with always selecting the best performing algorithm.

In a random selection method, the selector chooses one of the ten classifiers to use for each of the datasets. Statistically, each time this is done there is a 10% chance that it chooses the right classifier. This would result in 1.8 correctly predicted classifiers.

The other simple method is a bit less obvious, as "the best performing algorithm" is usually not explicitly defined by a single metric. This could for instance be defined as the algorithm which has, on average over all datasets, the highest predictive accuracy. Another way to define this term is to see which classifier is the best performing classifier on the most datasets.

Classifier	Average performance	Number of datasets
AdaBoost	0.8281490	1
Bernoulli NB	0.7655685	1
Decision Tree	0.8200023	1
Extra Trees	0.8503831	4
Gradient Boosting	0.8646888	10
kNeighbors	0.8376566	0
Multi-Layer Perceptron	0.8672565	18
Random Forest	0.8523825	0
Stochastic Gradient Descent	0.8186016	0
Support Vector Machine	0.8538678	7

Table 4: The performance of the 10 classifiers in two metrics: the average performance (measured in predictive accuracy) of each algorithm over all datasets, and the number of datasets for which the performance of the classifier was the best compared to the other classifiers. This information was gathered from the trainingset.

When we look at Table 4, we see that the Multi-Layer Perceptron performs the best. Although the difference on the average performance is small, the number of datasets the Multi-Layer Perceptron

performed the best on is almost twice that of the nearest competitor.

When we predict this best algorithm for every instance of the testset, we can see in Figure 13 that it is correct in 8 of the 18 cases. This is 44%, compared to the 55% score that our model predicted right and the 10% prediction score that is accomplished by using a random selection method.

6 Conclusions and Further Research

In this thesis, we have built a library for an OpenML-connector in C++. This connector can help to improve the number of reproducible studies. This library consists of the framework to perform and process API calls and part of the functionality available on the OpenML REST API. We then performed a case study to test the functionality currently available in the library.

After we built the library, we performed a case study that consisted of an algorithm selection problem. We compared the results of our model with both random selection and with the best performing algorithm. We found that our model performed better than both these methods, with our model predicting 55% of the algorithms and the baseline methods predicting 10% and 44%. This again proves the idea and practical feasibility of algorithm selection. As such, this practice should continue and possibly be expanded and improved upon

Although our model performed better than the two baseline methods, there is still quite some room for improvement. This improvement could for instance be achieved with better training methods, such as 10-fold cross validation or leave one out cross validation.

During the case study, the connector performed as expected and performed its tasks well. The framework works and can easily be expanded upon. This is important as much functionality is not yet implemented in the connector. This missing functionality also means that not much functionality could be tested.

Future work on this topic could include the addition of further functionality to the connector. This would greatly enhance the possibilities for potential experiments. Another possible enhancement would be to integrate the SHARK and mlpack packages and the connector. This could result in easier use and some highly automated systems.

References

[ASR+20] Edesio Alcobaça, Felipe Siqueira, Adriano Rivolli, Luís P. F. Garcia, Jefferson T. Oliva, and André C. P. L. F. de Carvalho. MFE: Towards reproducible meta-feature extraction. *Journal of Machine Learning Research*, 21:1–5, 2020.

[BCF+17] Bernd Bischl, Giuseppe Casalicchio, Matthias Feurer, Frank Hutter, Michel Lang, Rafael G. Mantovani, Jan N. van Rijn, and Joaquin Vanschoren. OpenML benchmarking suites. *arXiv e-prints*, page arXiv:1708.03731, August 2017.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[BLK+16] Bernd Bischl, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Erich Studerus, Giuseppe Casalicchio, and Zachary M. Jones. MLR: Machine learning in R. *Journal of Machine Learning Research*, 17(170):1–5, 2016.

[Bot98] Léon Bottou. Online algorithms and stochastic approximations. In David Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998. revised, oct 2012.

[Bre01] Leo Breiman. Random Forests. *Machine Learning*, 45:5–32, 2001.

[CBL+17] Giuseppe Casalicchio, Jakob Bossek, Michel Lang, Dominik Kirchhoff, Pascal Kerschke, Benjamin Hofner, Heidi Seibold, Joaquin Vanschoren, and Bernd Bischl. OpenML: An R package to connect to the machine learning platform OpenML. *arXiv e-prints*, page arXiv:1701.01293, January 2017.

[CEL+18] Ryan R. Curtin, Marcus Edel, Mikhail Lozhnikov, Yannis Mentekidis, Sumedh Ghaisas, and Shangtong Zhang. Mlpack 3: A fast, flexible machine learning library. *Journal of Open Source Software*, 3(26):726, 2018.

[CH67] Thomas M. Cover and Peter E. Hart. Nearest neighbor pattern classification. *IEEE Trans. Inf. Theory*, 13:21–27, 1967.

[con15] Conan – Reference. https://conan.io/index.html, 2015. [Online; accessed March-2019].

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. In *Machine Learning*, pages 273–297, 1995.

[Fri01] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.

[FS97] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119 – 139, 1997.

[FvK+19] Matthias Feurer, Jan N. van Rijn, Arlind Kadra, Pieter Gijsbers, Neeratyoy Mallik, Sahithya Ravi, Andreas Müller, Joaquin Vanschoren, and Frank Hutter. OpenML-Python: An extensible Python API for OpenML. *arXiv e-prints*, page arXiv:1911.02490, November 2019.

[GC08] C. Giraud-Carrier. Metalearning – A tutorial. https://pdfs.semanticscholar.org/54ac/a33d66ba256ff96ebd12b7016dd2d6d137c1.pdf, 2008. [Online; accessed December-2019].

[GEW06] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Mach Learn*, 63:3–42, 2006.

[IHMG08] Christian Igel, Verena Heidrich-Meisner, and Tobias Glasmachers. Shark. *Journal of Machine Learning Research*, 9:993–996, 2008.

[JSG19] Hadi S. Jomaa, Lars Schmidt-Thieme, and Josif Grabocka. Dataset2Vec: Learning dataset meta-features. *arXiv e-prints*, page arXiv:1905.11063, May 2019.

[Kal06] Marcin Kalicinski. RapidXML – Reference. http://rapidxml.sourceforge.net/index.htm, 2006. [Online; accessed April-2019].

[MN98] Andrew McCallum and Kamal Nigam. A comparison of event models for naive bayes text classification. In *AAAI 1998*, 1998.

[NR09] Peter Norvig and Stuart Russell. *Artificial Intelligence: A Modern Approach*. Pearson, 3rd edition, 2009.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[RGS+18] Adriano Rivolli, Luís P. F. Garcia, Carlos Soares, Joaquin Vanschoren, and André C. P. L. F. de Carvalho. Characterizing classification datasets: A study of meta-features for meta-learning. *arXiv e-prints*, page arXiv:1808.10406, August 2018.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and R. J. Williams. "learning internal representations by error propagation". In *Parallel distributed processing: Explorations in the microstructure of cognition*, volume 1: Foundations. MIT Press, 1986.

[RM05] Lior Rokach and Oded Maimon. Decision trees. *The Data Mining and Knowledge Discovery Handbook*, 6:165–192, 01 2005.

[Ste18] Daniel Stenberg. Everything curl. https://ec.haxx.se, 2018. [Online; accessed March-2019].

[VvRBT13] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. OpenML: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

[WFHP16] I.H. Witten, Eibe Frank, Mark A. Hall, and Chris J. Pal. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 4th edition, 2016.