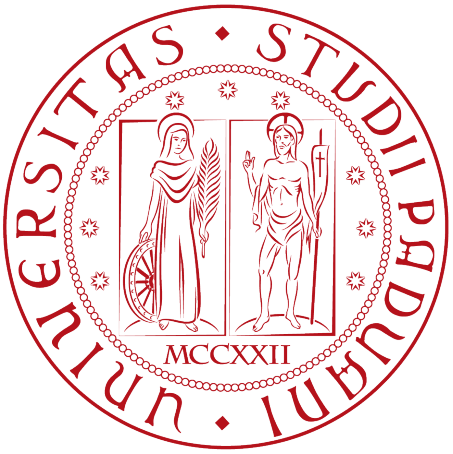
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea Magistrale in Informatica



A graphical data analysis tool for dataset enhancement and preprocessing

Master's Thesis

Supervisor
Prof. Silvia Crafa

Co-supervisor
Prof. Barbara Di Camillo

Graduate student
Alessandro Zangari

Academic Year 2019-2020

Abstract

The success of data-driven approaches, as well of data analysis methods, heavily depends on the quality of the data being used. Hence, these techniques rely on the application of specific procedures for cleaning, organizing, and preparing data for processing.
This document describes the development of DataMole, a new tool written in Python, equipped with a Qt-based graphical interface, that can support researchers during data exploration and preprocessing activities. Data transformation pipelines can be defined and executed within a simple, user-friendly graphical environment, effectively providing an intuitive approach to data manipulation. The tool also embeds functionalities for data visualisation through interactive plots, like scatterplots and line charts, and provides a specific feature for the extraction of time series from longitudinal datasets.
DataMole is open source and released for free usage under the terms of the GNU General Public License (GPL). The first version of the tool will be available on GitHub, along with the technical documentation, a developer manual and a user guide. Drafts of the manuals have been attached to this document in appendix.

Acknowledgements

First I want to thank professors Silvia Crafa and Barbara Di Camillo, my thesis tutors, for their support and encouragement throughout the development process and for their patience in reviewing this thesis.

A special thanks to the whole research team at the Department of Information Engineering that made this project possible. Your ideas and feedback has been invaluable.

I'd also like to thank professor Fabio Aiolli and Mirko Polato for their enthusiasm in sharing their advices and suggestions about the tool and its possible applications.

Finally, I want to express my gratitude to my parents and my friends for their continuous support and the great moments we shared during these years.

Padova, September 2020

Alessandro Zangari

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine learning datasets often require some manipulation and polishing before experts can actually use them in their algorithms. The quality of training data can have a considerable impact on the algorithm output, so time should be spent to clean and regularise the data, in order to remove all noise that makes the data unsuitable for machine processing. This is particularly important with datasets containing human supplied information, like the ones gathered in polls, population studies, medical records, etc. Additionally, data can contain much more information than what is reasonable to keep; hence, field-experts evaluation may be useful to discard all those data features that are not relevant for the task at hand, and to assess their correctness and value. Experts may also decide to improve and transform the data, a task-dependent activity that commonly involves some preprocessing transformations, like feature engineering, data encoding and scaling, in order to synthesise new, more expressive attributes and give the data a more convenient representation. Data visualisation can help as well, through informative statistics and charts, like scatterplots and heatmaps, enabling analysts to get a better understanding of the meaning of the data and can give some hints on how to enhance them.
This important groundwork can be tackled with any programming language, though some of them are traditionally more used for these tasks, and benefit from years of contributions from their communities that make them more comfortable to use for data-wrangling operations. Python and R comes with a considerably long list of data analysis packages which create a data-friendly ecosystem suitable for almost every possible need. Still, technical abilities and programming skills are required to be able to use these tools fluently, and a significant amount of time can be spent by anyone approaching them for the first time. Thus, it is natural to wonder how a software with a graphical interface could help users in their exploratory work on datasets: such a tool should allow its users to transform and manipulate the data, enabling them to keep track of their progress, through statistics and visualisation features. This kind of tool can effectively support the work of interdisciplinary teams of experts, even non-programmers, providing them a set of powerful features to make data preparation a less time-consuming task and helping novice users to grasp the meaning

and importance of dataset preprocessing. Additionally it is of critical importance that preprocessing steps can be reproduced on different datasets: this allows to apply the same transformations on similar datasets, and ensures reproducibility of the whole procedure.
This thesis describes *DataMole*, a user-friendly graphical tool created to help its users to perform some of the aforementioned operations. The tool is inspired from Weka, a machine learning workbench with graphical interface with similar functionalities, but it is simpler to use (although more limited) and includes specific features to simplify exploration of longitudinal datasets. Additionally it is developed in Python, to take advantage of the many available packages, and internally uses Pandas and Scikit-learn for data manipulation. Besides, Qt for Python is used to create the graphical interface.
When exploring an unknown dataset, experts may be interested in having a look at the data features, seeing their correlation, and then applying some cleaning operations, like removing unwanted attributes, changing data types or dealing with NaN values. In other situations the dataset is already known, and a way to rapidly apply a set of predefined transformations should be provided. DataMole supports these two operative modes separately: it is possible to apply single transformations to individual datasets while looking at the data, or the user can apply a complete pipeline of transformations, that can be defined easily by dragging and dropping operations and connecting them to create a graph. Pipelines can also be saved and later applied to other data: this feature provides support for transparent data manipulation and boosts reproducibility by making transformations easily replicable.
To get a sense of data before and after transformations, chart visualisation features are provided with frequency histogram, scatterplot and time series line chart.
Finally, the tool integrates some specific features for time series extraction and visualisation, tailored for usage with longitudinal datasets. These datasets track the same sample of subjects through time, periodically monitoring some parameters of interest. The tool provides a way to visualise such features over time, through the creation of a normalised dataset which can be interpreted as a time series.
DataMole focuses on helping non-programmers to transform and clean their data: the intuitive graphical interface guides the user in choosing the right transformation, by offering a detailed description of how it operates and providing a simple widget to configure it with the required parameters, that are additionally validated in order to inform the user of any wrong combination of arguments.
The design of a graphical interface to create lazy pipelines of transformations, required the development of a computational graph infrastructure to allow the propagation of information through the graph and support multithreaded execution of every node.
Finally, many architecture design decisions were made in order to favour the embedding of new features. Hence, a developer guide is included in the appendix of this document to simplify software extension.

1.1 Project objectives

DataMole was the result of a collaboration between the Department of Mathematics and the Department of Information Engineering of the University of Padua. Its primary goal was the creation of an open source graphical desktop application capable of extracting and polishing information contained in medium-sized biomedical datasets, including longitudinal datasets. These datasets typically contain a huge amount of features that need to be filtered, extracted, cleaned and prepared for usage in data analysis tasks.
A list of the main *functional requirements* is provided below.

- **Preprocessing capabilities**: the tool should allow the application of common preprocessing operations, like management of missing values, feature discretization, scaling, standardization and simple dataset manipulations, like the ability to add and remove or duplicate columns;
- **Visualisation features**: the tool should be able to help data exploration through the presentation of informative statistics about the dataset and the creation of charts, like scatterplots and histograms;
- **Transformation pipeline**: the ability to create preprocessing pipelines with a graphical interface: this approach is often used for data manipulation, and is easy to visualise, understand and convenient for sharing;
- **Interoperability**: the ability to import existing datasets into DataMole and export them in order to continue working on them with different tools;
- **Transformations tracking**: it should be always possible to understand which transformations were applied to a particular dataset. This is important to document the applied transformations through a logging functionality;
- **Support for multiple datasets**: the ability to load in the software more than one dataset and work on them in parallel;
- **Support for longitudinal data**: since longitudinal datasets contain temporal information, the tool should have the ability to plot feature values over a time axis. The structure of longitudinal datasets is covered in the next section, with some examples.

Some additional *desirable* requirements are listed below. They describe possible enhancements and features that were not considered indispensable.

- **Pipeline export**: the possibility to export existing pipelines for sharing and to keep a documentation of the operations applied to a dataset;
- **Support for train/test set**: the tool should allow to fit pipelines on a training dataset and apply them (without re-fitting) to a test set; this kind of support is often required when processing data for machine learning;

- **Scalability**: the tool should support out-of-memory dataset and out-of-core computations, in order to support big datasets that do not fit in the machine main memory;
- **Undoable transformations**: the tool should give the possibility to revert the last applied operation;
- **Customised functions**: the possibility to define functions and apply them to a dataset.

Finally, the main *quality requirements* for DataMole are the following:

- **Extensibility**: the software should be designed in a way to favour the addition of new features;
- **Documentation**: implementation and architectural choices should be documented in a *developer manual* and the program features and their internal working should be described in a *user manual*;
- **Open source**: the software should preferably be released with a license that allows to continue its development; hence, any third party technology embedded in this software should be chosen appropriately, in order to fulfil this requirement.

1.1.1 Description of longitudinal datasets

Longitudinal datasets contain data from repeated observations of the same sample (e.g. people) over a period of time. Thus some variables of interest of the observed sample are measured on a continuous basis. Specifically, the team behind this project is working with the *English Longitudinal Study of Ageing* (ELSA) [1] and the *Health and Retirement Study* (HRS) [9]. The next sections present these two datasets, that will be later used to give some examples of DataMole features.

1.1.1.1 The ELSA dataset

The *English Longitudinal Study of Ageing* was started 2002 by the National Institute on Aging (NIA) and involves a sample of the English population aged 50 and older.
Every two years, the participants are asked to complete a core self-completion questionnaire covering questions such as well-being, relationships, alcohol consumption, household and socio-economic status. Hence this longitudinal dataset is composed of data gathered during successive *waves* of interviews, carried out every two years. Currently the study made available data from 9 waves, with the last one completed in 2020. Additionally, every 4 years, participants are visited from qualified nurses and various physical examination and performance data are obtained, while some biological samples are collected for analysis [6]. Waves may not contain data from every participant, since they may refuse to be interviewed or they may not be available when required. Similarly, while completing the questionnaire, participants can always skip questions if they prefer not to answer. Thus data is not homogeneous and contains

Description	Waves			
	2	3	4	5
Chronic lung disease ever diagnosed	hedibw1	dhediblu	hediblu	hediblu
Whether respondent felt sad much of the time during the past week	PScedG	pscedg	pscedg	pscedg
Level of physical activity in main job	wpjact	wpjact	wpjact	wpjact
Amount of income from work in the last year	wpwlyy	wpwlyy	wpwlyy	wpwlyy
Average monthly income from business in the last 12 months	wpbima	wpbima	WpBIma	wpbima

Table 1.1: Descriptions of variables from the ELSA dataset with their name in 4 waves

a lot of missing values, that may be missing for several reasons, so different codes are used to indicate these possibilities.
Every wave of data can be downloaded as a separate file from the UK Data Service online portal, along with the data documentation needed to interpret it: for instance a data dictionary is used to associate every variable with its description, its type and possible values. Every file contains thousands of variables which corresponds to the answers given to the ELSA questionnaire or, in case of the nurse visit datasets, to the physical parameters measured during the visits. Every variable is placed on a different column named with a code that is used to identify the feature. Besides every participant has a cross-wave identifier that allows to recognise the same subject in different waves. An example of three variables contained in the ELSA dataset, with their description is shown in Table 1.1. Notice that variables names are not always consistent through waves. Additionally some waves may contain slightly different questions in place of older ones, or others may be deleted entirely starting from a wave.

1.1.1.2 The HRS dataset

The *Health and Retirement Study* is a longitudinal panel study that surveys a representative sample of approximately 20000 people in America. It is conducted by the University of Michigan and founded by the National Institute on Aging and the Social Security Administration. It contains 16 waves from 1992 to 2018 and the participants are interviewed every two years. These interviews cover a wide range of topics, including family structure, health conditions, employment history and much more. Similarly to ELSA, some physical parameters (e.g. blood pressure, waist and hip circumference, lung function, grip strength, blood parameters) are measured every 4

years from a selected subsample of participants during specific examinations.
The public dataset can be downloaded for usage with statistical software like Stata or SPSS, and it can be manually converted to a plain `CSV` file. Its structure is very similar to ELSA, where columns represent variables and an unique identifier is provided to recognise subjects through different waves.

1.1.1.3 Critical issues in longitudinal datasets

Longitudinal datasets like ELSA and HRS contain thousands of variables. DataMole allows searching through the dataset and to extract specific subsets of columns, since studies that use these datasets are typically interested in just a few variables.
These data are in large part self-reported by participants and often contain a lot of missing information. For example the *wpbima* attribute from ELSA listed in Table 1.1 contains more than 97% of missing values in several waves. Sometimes this happens because respondents choose not to share some more sensible information (like the average monthly income) or because some questionnaire sections are skipped under certain conditions (e.g. the respondents does not know the answer, the question is not applicable to his/her category, etc.). To communicate these conditions special codes are used. Some variables may also be dependent on other variables: for example questions about previous employment are skipped if the respondent never had a job, thus all such questions will be considered *not applicable* and marked with a specific missing code. DataMole embeds a transformation that can be used to replace values, so that these codes can be properly treated as missing values.
Values can also be inconsistent across different waves: for example a subject may declare that a diagnosis of a lung disease has been reported to him, and dispute this fact in a following wave. This may happen if the diagnosis has been made and later proven incorrect, or it may result from a simple mistake. DataMole does not have a specific feature to manage these inconsistencies, because the correct way to deal with them differs from case to case, and it may involve some ad-hoc transformations that need to be done outside of DataMole.

1.2 DataMole overview

DataMole is a data manipulation software suitable for dataset exploration and data cleaning, that can be used through a graphical interface. It is written in Python and uses the Pandas library and specifically its *dataframe* data structure to open and manipulate datasets. Its features include:

- Import and export datasets in `CSV` format and with `pickle`;
- Visualise the dataframe in a table;
- Display a list of dataframe columns with their name and type and search for specific attributes by name;
- Visualise statistics for every column, and an histogram to see value distribution;

- Apply transformations to specific attributes, like scaling, one-hot encoding, and also manage datasets by renaming, dropping and duplicating columns;
- Load more than one dataset and switch between open datasets to explore and compare them;
- Side by side view for open datasets for fast comparisons;
- Define pipelines of transformations with the possibility of exporting and importing them;
- Draw bivariate scatterplots and time series plots;
- Automatically log every applied operation along with its configuration for documentation purposes;
- Easy-to-use graphical interface: every data transformation can be parametrised using simple widgets and pipelines can be defined with drag-and-drop.

1.2.1 Related work

Most software for data preparation tasks is packaged inside libraries or frameworks usable within programming languages. However there are very few tools that allow to do this from a graphical interface, not counting commercial software. Two of them are briefly described here:

- *Weka*, acronym of Waikato Environment for Knowledge Analysis, is an open source software written in Java, developed by the University of Waikato (New Zealand) that provides a collection of machine learning and preprocessing algorithms that can be applied from a graphical interface [8]. Weka is a complete machine learning suite, and offers much more than just data preparation features;
- *DataPreparator* is a free-to-use graphical software also developed in Java, designed to assist with common tasks of data preparation for data analysis, and includes features for data cleaning, preprocessing and visualisation [2].

1.2.1.1 Comparison with DataMole

Both the above software contain more features with respect to DataMole. As already stated, Weka is a graphical machine learning suite capable of defining entire machine learning pipelines, complete of preprocessing steps, training, validation and testing. It additionally supports community developed extensions that further enrich the list of available features. DataMole is more limited, but offers these functionalities that Weka does not provide:

- Better *CSV support*: Weka is very sensitive to `CSV` formatting errors, which unfortunately are quite common in the already cited longitudinal datasets; on the other hand, Pandas handles well even wrong-formatted `CSV` files;

- *Operation log*: DataMole keeps a trace of every applied operation, which can be useful to document an experiment;
- *Multiple datasets*: more than one dataset can be loaded in memory and the user can easily switch between them; a side by side view of different datasets is also available for comparisons;
- *Column search*: allows searching through dataset columns by name or regular expression; this is particularly convenient when the dataset contains thousands of columns, and provides a way to quickly filter attributes and select them when configuring a transformation;
- *Time series visualisation* for longitudinal datasets: time series can be extracted from this specific kind of datasets and plotted; this feature is described with an example in the next chapter;
- *Interoperability with Python*: since DataMole is written in Python, it allows exporting datasets in *pickle*, the serialised binary format for Python; this is extremely useful to continue working on a dataset outside of DataMole, without having to parse the dataset again.

DataPreparator does not have all Weka functionalities, but still provides advanced features, like support of data streaming, more visualisation features and supports the creation of fitted pipelines that can be exported and applied to a test set. The main drawback of this tool is that it is not open source, hence not extendable, and it is not maintained: the latest version was released in 2013. On the other hand DataMole is open source and includes some convenient extension mechanisms described in the developer manual (in Appendix A) that provides a way to extend it with additional features.

1.2.1.2 Extension of existing tools

Before starting to develop DataMole, some time was spent exploring Weka and DataPreparator, in order to determine which features they provide and how they could be improved. In addition, we considered the possibility of extending Weka. This software was designed to allow extension: a dedicated section is present in the Weka user guide and new plug-in extensions can be published and distributed through an official repository. Weka is composed of many panels that can be launched from a main window and some of these support the embedding of new features: for example, in the *Explorer* panel, extensions with new functionalities can be placed on new tabs, while the *Knowledge Flow*, which allows the definition of complete machine learning pipelines, can be extended with new customised *steps* (which are operations on the datasets).
The Weka Explorer panel was one of the main inspirations for the earlier concepts of this tool, so we considered adding the missing functionalities (like time series visualisation) directly to it. However some features would have required a reimplementation of

some part of the software, and not just an extension. For instance, Weka does not allow to open more datasets at the same time in the Explorer, and we wanted to provide this functionality to better support working with longitudinal datasets that are often split in many different files. More importantly it had issues opening the HRS or ELSA datasets from `CSV` files, probably because of some formatting errors or unsupported characters.
Eventually we decided to develop our own independent tool, with the advantage of having more control over design choices and the objective of starting a project that could be improved over time.

Chapter 2

Design of the data manipulation tool

This chapter briefly introduces the software interface and discusses the main considerations involved in its design.

2.1 GUI description

DataMole is centred around three panels, accessible in three different tabs:

- The *Attribute panel* for dataset exploration;
- The *View panel* for charts creation;
- The *Flow panel* to build preprocessing pipelines.

Each of these is described in the next sections, along with its features.
The left side of every panel contains the *workbench*, which allows switching between loaded datasets. Above the workbench, a widget shows some general information about the selected dataset, like its name, column and row numbers. Fig. 2.1 shows these two widgets marked with letters *(W)* and *(G)* respectively. Additionally various dataset transformations can be applied from this side-panel, like type conversions, feature discretization, one-hot encoding and many more. A detailed list of the available transformations is in §2.2.
Many screenshots and examples of DataMole features described in this chapter are based on the *iris* dataset. This is a very simple and well known dataset which measures the length and the width of both sepals and petals of 150 iris flowers, belonging to 3 different species: *versicolor*, *virginica* and *setosa*. More information on this dataset can be found in the original paper, available at [7].

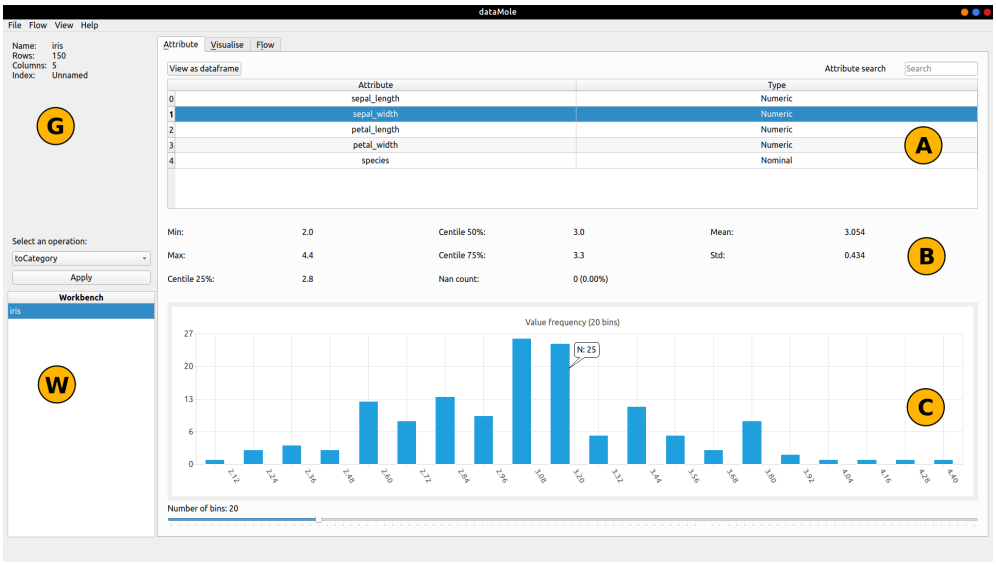


Figure 2.1: Exploration of the *iris* dataset in the *Attribute panel*: datasets can be switched in the workbench *(W)* and information for every attribute in the selected dataset are shown in sections *(B)* and *(C)*. The attribute table *(A)* allows to change the selected attribute.

2.1.1 Dataset exploration

DataMole provides the basic functions to explore a dataset and understand its content through some statistics within the *Attribute panel*. The visualisation of the *iris* dataset from this panel is shown in Fig. 2.1.
To understand a dataset the user needs to get an overview of the content of each column. To do this, the panel is organised in three sections:

- The *attribute table* keeps a list of every column in the dataset with its name and type (*(A)* in Fig. 2.1);
- The *statistics panel* shows some statistical information for the selected attribute *(B)*;
- An *histogram* displays the frequency of values within the selected column *(C)*.

DataMole uses Pandas to load and hold the dataset in memory, which automatically infers some types when it parses a file. In DataMole five data types are supported:

- *numeric*: represents both integers and floats;
- *datetime*: used to encode dates and timestamps;
- *categorical*: used to encode non-numeric attributes with a small set of distinct values. A categorical attribute can be:
  - *nominal*: if no order relation is defined between categories;
  - *ordinal*: if categories can be ordered.

- *string*: used for textual columns.

In the example in Fig. 2.1, the *species* column, which declares the species for every iris in the dataset, has been encoded as a *nominal* attribute, since it contains only 3 values with no order relation defined between them.
The type of information shown in the *statistics panel* depends on the attribute type: attribute statistics for numeric attributes include the minimum, maximum, average value, standard deviation and percentiles. For non-numeric columns this widget shows the number of distinct values along with the most frequent one, as well as the minimum and maximum date for datetime attributes. Additionally the percentage of missing (NaN) values is always shown.
For non-numeric attributes, the *histogram* shows the frequency of every distinct value. On the other hand, continuous-valued attributes (i.e. of type either numeric or datetime) are grouped in equal-size intervals, and their number can be changed by moving the slider on the bottom.

2.1.2 Plotting data

Aside from the histogram, which is shown in the *Attribute panel*, every other visualisation feature is included in the *View panel*. It allows to create informative charts from the data. For now, it is possible to generate two type of charts:

- *Scatterplot*;
- *Line chart*, only for time series.

Double clicking on a chart opens it inside an independent window and allows to save it as an image.
Even though this panel currently allows to create only two type of charts, its name was set on purpose: it is meant to contain a collection of tools to *visualise* various type of dataset information. Currently only charts can be created but the panel can be easily extended with new features, which may possibly include not only charts.

2.1.2.1 Scatterplot matrix

The *scatterplot matrix* is often used to visualise the correlation between pairs of attributes and to understand how well they can discriminate a target feature. For example Fig. 2.2 shows a scatterplot matrix on the 4 attributes of the *iris* dataset. Every point is coloured with respect to its target category, which is set to the *species* attribute. The widget on the right side of the matrix *(A)* allows to select which attributes to include and to select the target feature. Double clicking a chart opens it in a window which additionally displays the legend and allows to zoom, pan and resize the chart. Single charts can be saved as images, like the one in Fig. 2.3.

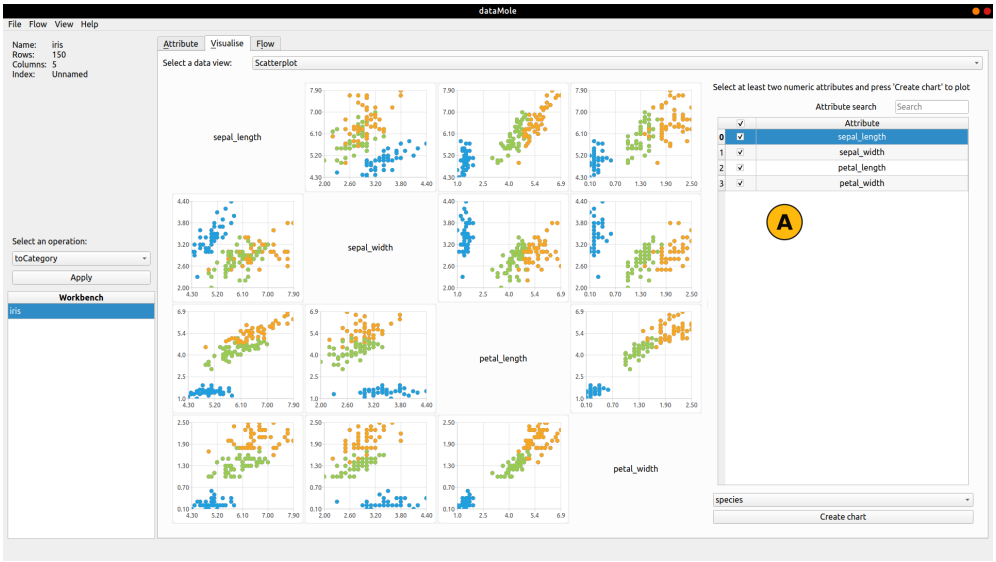


Figure 2.2: The *View panel* showing a scatterplot matrix on the *iris* dataset. Every point is an entry of the dataset and its colour indicates the flower category, that can be selected in the right panel *(A)*.

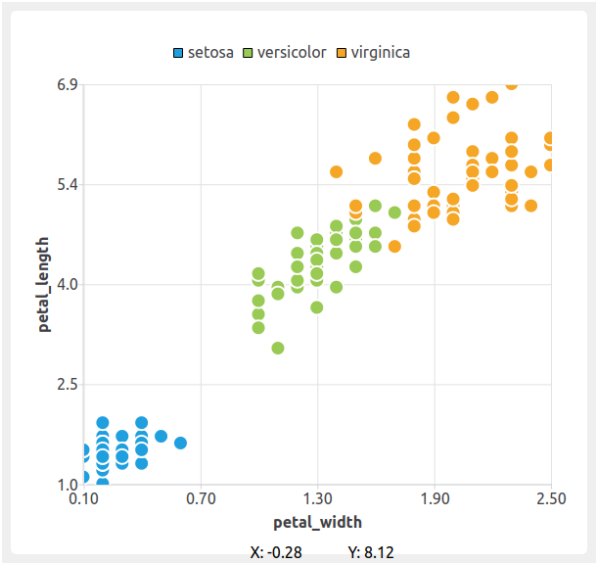


Figure 2.3: A `PNG` image of a scatterplot from the *iris* dataset

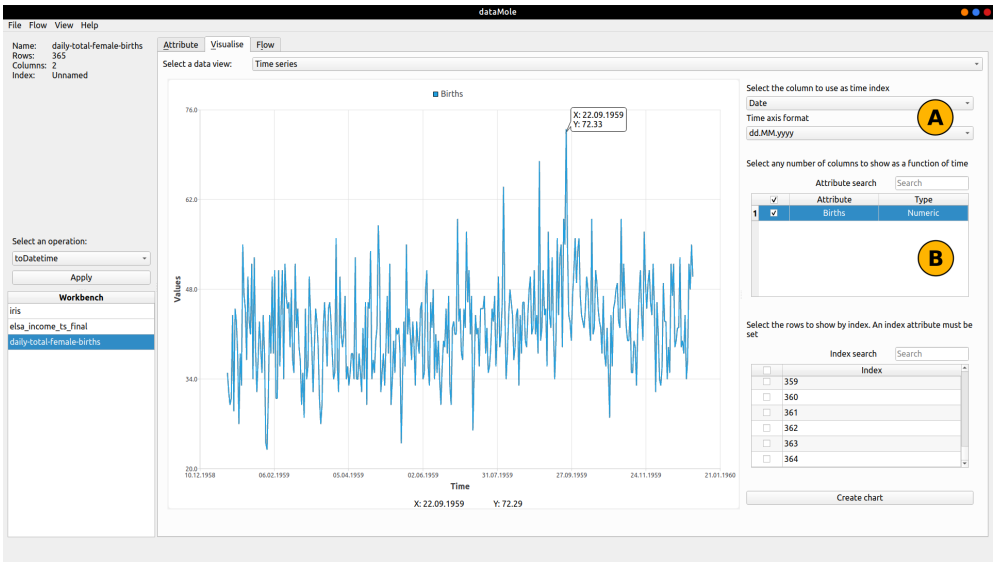


Figure 2.4: The time series line chart showing the daily number of female births in California during year 1959. Combo boxes in *(A)* allow to select the time axis and change its format and the time series values can be selected in *(B)*.

2.1.2.2 Time series plot

Datasets with feature values spread out over a time axis can be visualised in a line chart that displays data over time. Fig. 2.4 shows the daily number of female births in California during year 1959, taken from a public dataset found on Kaggle [3]. Every entry in this dataset contains the time label (day of the year) of type *datetime* and the number of births. The widget on the right side of the chart allows to configure it: the attribute to use as time axis can be chosen in *(A)*, while the dependent variables to plot over time can be selected in *(B)*.

2.1.3 Defining pipelines of transformations

Through the *Flow panel* its possible to create pipelines of transformations: a pipeline defines a *flow* of operations on the data where the output of a node is the input of the next connected operation. Fig. 2.5 shows a screenshot of a simple pipeline, marked with *(F)*. Transformations can be added by dropping them from the list placed on the left side of the window *(L)*. Additionally, most transformations must be configured before the pipeline can be executed, and this is covered later, in §2.3.4.
One of the advantage of the approach to dataset transformation proposed in this panel is that entire pipelines can be exported and imported to be used with other datasets. All the available data transformations are described in §2.2.

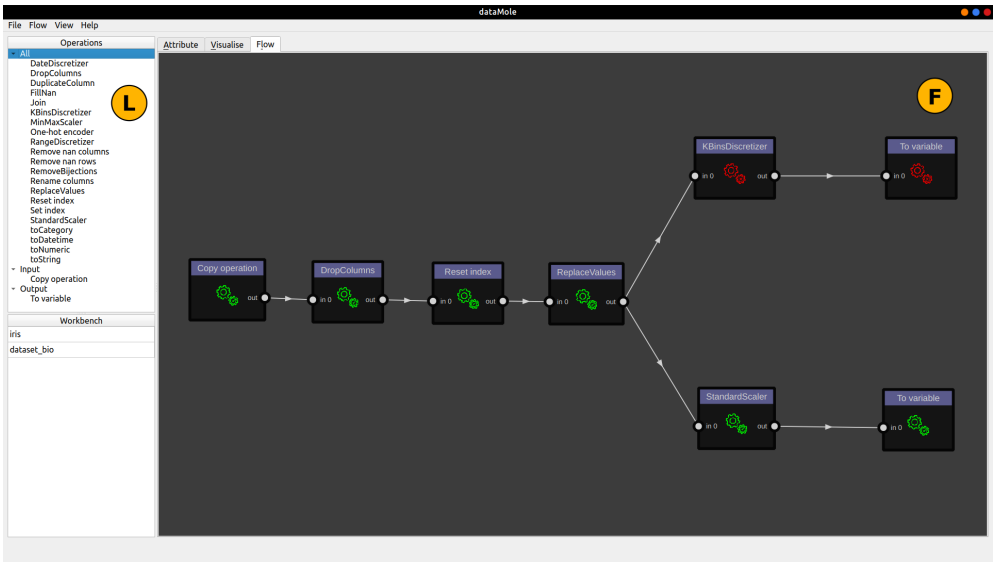


Figure 2.5: A data pipeline in the *Flow panel*: dataset is read using the *Copy operation* and the pipeline result is written on the workbench using the *To variable* operation. New operations can be added dragging them from the left panel *(L)*.

2.1.4 Inspiration for GUI design

Some features of DataMole as well as its graphical interface has been inspired by existing software, in particular Weka and DataPreparator, already described in the introductory chapter.
The main *Attribute panel* has been inspired by the Weka *Explorer panel*. Weka presents data in a similar way, with statistics and an histogram. It also allows to apply transformations to the data by selecting the operation (called *filter*) and configuring its arguments. Differently from DataMole, the Weka Explorer does not allow to keep open more than one dataset at a time and to search for specific attributes, which can instead be useful when dealing with many columns. In fact, this is particularly important to work with the longitudinal datasets that contain thousands of columns, and are usually split into many files.
The possibility to create a pipeline-based interactive mode, available in the *Flow panel*, has been inspired by both the Weka *Knowledge Flow* and by a similar feature of DataPreparator. The Weka pipeline is more advanced, since it allows to define complete machine learning pipelines and supports data streaming. Similarly, DataPreparator data processing capabilities are centred around the creation of an operator tree where each node is a different operation that transforms the dataset. The interaction with DataPreparator operator tree is quite different from DataMole, since every node must be configured and executed manually before its successors can be run.
The implementation of the DataMole pipeline is described in chapter 3.

2.2 Data manipulation features

One of the DataMole core features is the ability to apply *transformations* to the dataset. The following transformations are available:

- *Type conversions*: as explained in §2.1.1 types are automatically inferred when a file is parsed, but the program allows to explicitly set types and convert between them if needed;
- *Dataset join*: it is possible to join pairs of datasets on specific columns or alternatively on index columns. Inner, outer, left and right SQL-like join is supported;
- *Missing values management*: NaN values can be imputed with different strategies, for example by replacing them with the column average value, with a specific value or with the last valid value. Additionally rows or columns with a high number of missing values can be removed;
- *Indexing*: sometimes it is useful to set one or more attributes as indices of the dataset, for instance, before joining two datasets;
- *Scaling*: attributes can be scaled to a specified range with *min-max scaling* or standardised with respect to their mean and standard deviation;
- *Discretization*: continuous features (i.e. of type numeric or datetime) can be discretized into a variable number of bins with different strategies, including equal-sized bins, equal-frequency bins and manual range specification;
- *One-hot encoding*: nominal categorical features and string attributes can be one-hot encoded;
- *Cleaning operations*: attributes values can be replaced with different values, columns can be renamed, duplicated and dropped;
- *Time series extraction*: one of the requirements for this project was the ability to visualise the temporal information contained in longitudinal datasets. To be interpreted as a time series, this information must first be extracted with an operation designed for this purpose.

Most of these transformations are quite standard in the data science domain. On the other hand, the *time series extraction* feature was specifically designed to work with longitudinal datasets and its purpose is described with an example in the next section.

2.2.1 Extraction of a time series: an example

Date	Births
1959-01-01	35
1959-01-02	32
1959-01-03	30
⋮	⋮
1959-12-30	55
1959-12-31	50

Table 2.1: Some rows from the *daily-total-female-births* dataset

idauniq	wpwlyy	wpbima	...
121321	...	...	
121323	...	...	
121332	...	...	
⋮			

Table 2.2: Structure of a wave of the ELSA dataset

The dataset used to plot the time series in Fig. 2.4 is composed of 365 entries, each with a date (under column *Date*) and the number of female births on that day (column *Births*). A sample of this dataset is reported in Table 2.1. The *View panel* can understand dataset in this simple format: it is only necessary to convert the *Date* attribute to *datetime*, set it as the time axis and select the time-dependent attributes, which is only *Births* in this example.
Longitudinal dataset like ELSA or HRS are not so simple: first there is no attribute representative of the time axis, but rather this information is implicitly conveyed by putting the variables from different waves in distinct files, or, if they are all in the same file, by renaming the variables to reference the wave they belong to (e.g. *varA_wave1*, *varA_wave2*, etc.). Hence, with these datasets the temporal information must be made explicit, by defining a time axis and, with the user help, link every column of the datasets to their point in this axis. For instance the file containing ELSA data from wave 3 has the structure described in Table 2.2. For the purpose of this example, only 3 attributes are listed: the *idauniq* attribute contains the unique cross-wave identifier, *wpwlyy* is the amount of gross income gained from the respondent work at the end of the previous year and *wpbima* is the average monthly income from business during the previous 12 months (from the interview). Other waves follow the same scheme, even though variable names can sometimes change across different waves. Suppose we want to visualise how the respondent income changes from wave 2 to 8 and plot it as a time series: there is no immediate way to do it, and we have to manipulate this dataset and transform it into a time series that can be identified and used in the *View*

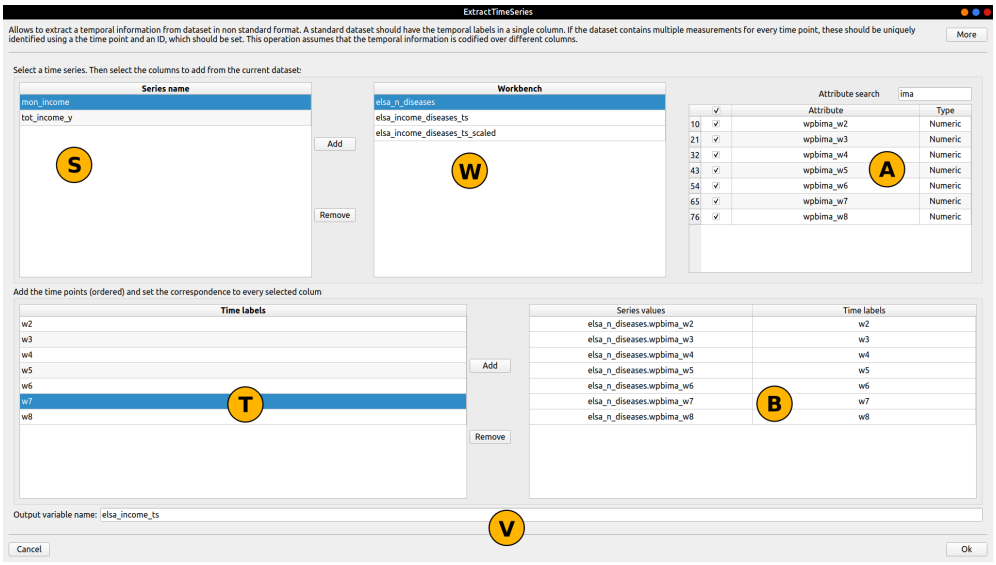


Figure 2.6: Extraction of time series from ELSA attribute *wpbima*

panel. The *time series extraction* operation is designed for this use case. Fig. 2.6 shows the widget used to configure the operation. In this case the relevant attributes (*wpbima* and *wpwlyy*) were previously extracted from their files and merged together in a single dataframe using the *join* operation. They were also renamed by appending the name of the waves they referred to (*_w2* for the second wave, *_w3* for the third and so on). This attributes contain a majority of NaN values: for the purpose of this example, they were filled with the per-column average, even though, in a real use case, they would probably be unusable.

In Fig. 2.6 two time series are defined and added to list *(S)*: *mon_income* for attribute *wpbima*, and *tot_income_y* for *wpwlyy*.

Then every time series is populated with its values: since we are interested in the average monthly income from the second to the eight wave, attributes from *wpbima_w2* to *wpbima_w8* are selected in table *(A)*. Here all attributes are taken from a single dataframe, but it is generally possible to select attributes from different dataframes by choosing the right ones in the workbench shown in section *(W)*.

The next step is the association of these attributes to a time point (the wave in this case). Thus, after defining an appropriate number of labels for the time axis in the bottom-left table *(T)*, we associate every relevant column to its time label in the bottom-right table *(B)*: here we are basically telling DataMole that *wpbima_w2* contains the values of the attribute *wpbima* for wave 2, *wpbima_w3* contains the values for wave 3 and so on. Series *tot_income_y* is built in the same way, by selecting the attributes *wpwlyy* from the various waves. Finally, a name for the dataset is set in *(V)* and the operation is started.

The result of this operation is a new dataset with the structure shown in Table 2.3. Every respondent is associated to 7 entries, each containing a distinct time label (*time* attribute) and the values of the *wpwlyy* and *wpbima* columns, respectively under

idauniq	time	tot_income_y	mon_income
121321	w2	...	...
	w3	...	...
	w4	...	...
	w5	...	...
	w6	...	...
	w7	...	...
	w8	...	...
121323	w2	...	...
	w3	...	...
	w4	...	...
	w5	...	...
	w6	...	...
	w7	...	...
	w8	...	...
121332	w2	...	...
	w3	...	...
	w4	...	...
	w5	...	...
	w6	...	...
	w7	...	...
	w8	...	...
⋮			

Table 2.3: Structure of the dataset generated with the extraction of two time series (*tot_income_y* and *mon_income*) from the ELSA dataset

columns *tot_income_y* and *mon_income*.
At this point it is possible to plot these attributes: you can see how the average monthly income and the yearly income changes for a single respondent (Fig. 2.7) or you can compare the same attribute (yearly income or monthly income) from two different respondents (Fig. 2.8). In both cases the table on the bottom (named *(C)* in Fig. 2.7) must be used to select the ids of the respondents to plot. This table allows to select by dataframe index, that must be set appropriately: in this example it is set on the respondent id. Additionally when an index is selected, the chart expects to find a time attribute and some time-dependent attributes to plot for every selected index, exactly like in Table 2.3. Hence table *(C)* should be used only to plot time series extracted with the *time series extraction* operation.

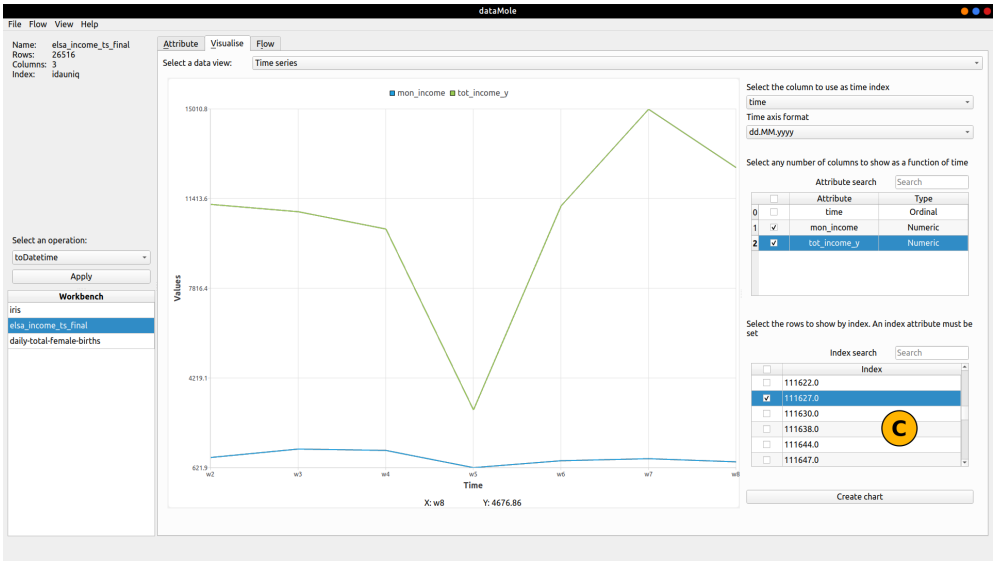


Figure 2.7: The average monthly income and yearly income of a single ELSA respondent measured in wave 2 to 8 (ELSA attributes *wpbima* and *wpwlyy*). The time labels for every wave are shown on the *Time* axis (horizontal), while the vertical axis shows the values of the two selected attributes.

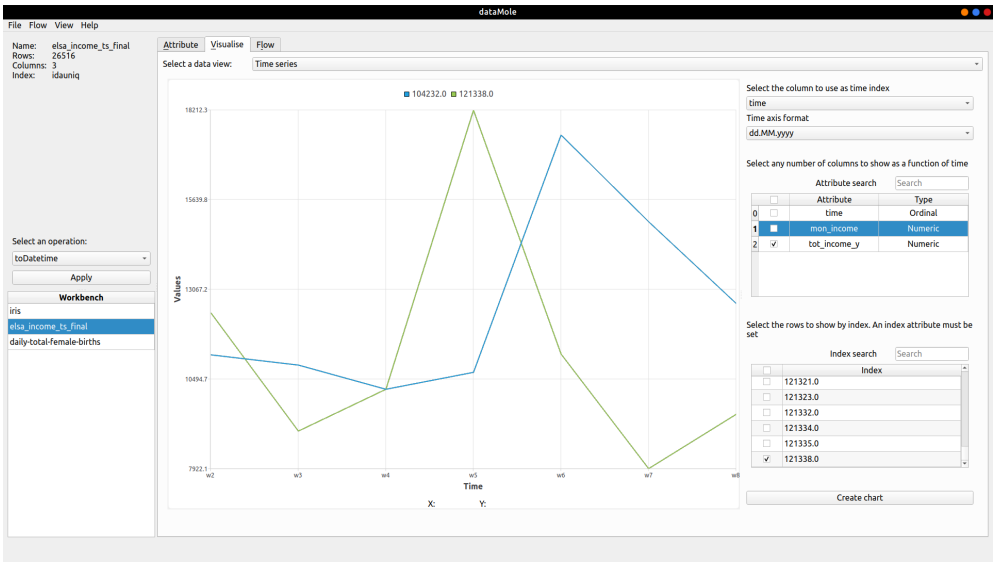


Figure 2.8: The yearly income of two ELSA respondents from wave 2 to 8 (ELSA attribute *wpwlyy*)

2.3 Conventions and other features

The following sections discuss some features that were added to improve DataMole user-friendliness, its transparency with respect to data manipulation and its interoperability with Python.

2.3.1 Treating missing values

Missing values are codified using NaN: this is a standard practice when working with preprocessing tools, like Pandas and Scikit-learn. This encoding for missing values is independent of the type of the column. The number of NaN values within a column is reported in the *statistics panel* available from the *Attribute* tab.
DataMole makes some choices about missing values that are sometimes different from how they are treated in Pandas and Scikit-learn. These packages do not always support computations with NaN values. For example the Scikit-learn discretization functions available in the current version raise an exception if any selected column contains NaN. In addition, Pandas does not always behave consistently with NaN values. As an example, if a numeric column with missing values is converted to string, every NaN in this column is replaced by the string "nan", that is no longer considered a missing value. However it still uses NaN values to encode empty strings when it parses non-numeric columns from a file.
This behaviour can be justified by the generality of this tools: they are used in a great variety of tasks by a large user-base and while their behaviour may be surprising in some situations, it may be desirable and expected in others. Additionally they offer an API for usage within programming languages, thus they expect constant supervision from a skilled user with programming background: if any exceptional condition is encountered (like the presence of NaN) they may stop executing and allow the user to personally deal with it, or they might expect the user to be aware of the problem, thus making a choice that may not always lead to the intended result.
On the other hand, this software treats missing values consistently: every defined transformation supports missing values and propagates them transparently. They are generally ignored and unaffected by every transformation, unless it is designed to explicitly treat them, for example to fill or remove them. For instance, the discretization feature provided in this software does not change the number of missing values, but rather processes only valid numeric values.

2.3.2 Types representation

When integer and real-valued columns are processed with DataMole they are marked as *numeric* and no distinction is made between integer and real numbers. As a consequence, for consistency, integers are always converted to real values when a new dataframe is created. Numeric columns are therefore treated as *continuous*. If a *discrete* encoding is required, the *categorical* type can be used instead.
One thing to keep in mind is that both *string* and *categorical* data types internally

represent values as strings. This may seem obvious, but Pandas supports an *object* type that can contain any Python type. Thus object columns may contain a mix of types, like integer, strings and boolean. Instead DataMole forces every column to contain a single type. This greatly simplifies the application of certain transformations that require the user to specify values found within the dataframe. For example, in order to substitute values the user must provide the value to replace as well as the replacement, and, to create a categorical column, the user can specify which categories to keep. However, if mixed type columns were supported, the software would not know how to parse these user-supplied values, since it would not know how they are represented inside every column.

2.3.3 Data transformation paradigms

There are two alternatives ways to apply transformations to the dataset:

1. Applying single operations in the *Attribute panel*;
2. Setting up a pipeline and executing it in the *Flow panel*.

The two approaches are not completely equal, since the first one does not allow applying operations that require multiple inputs, while the *Flow panel* can do that. As a consequence datasets join can only be done within the pipeline. Conversely, some operations may not be available in the graph. This is the case for the operation used to extract time series information from longitudinal datasets. Using it as a pipeline operation would not make sense, since it is only used to prepare data for visualisation in the *View* tab.
Aside from this two exceptions, every other operation can be applied from either panel. This may seem confusing, but these two panels are intended to be used in two different scenarios. In the first one the user is unsure about which operation to apply and wants to *try* various operations, while keeping an eye on the statistics to see how the dataset is changing. This may be the case when a new unknown dataset is imported, and the user wants to explore it.
On the other hand if the user knows in advance which transformations to apply or is interested in defining a pipeline he can apply to many different datasets (but with the same shape), then the *Flow panel* approach is more convenient. Pipelines can be exported in `pickle` format, which allows to serialize arbitrary Python objects.

2.3.4 Parametrised transformations

Most transformations described in §2.2 require to be configured before being applied. This ensures that every transformation is general enough to be useful in many different situations. Most operations can be applied to a specific subset of columns that must be selected by the user. Other options are more operation-specific: for example, missing values can be filled with many different strategies and min-max scaling requires the user to specify the scaling range.
Every operation that requires options from the user can be configured using an *options*

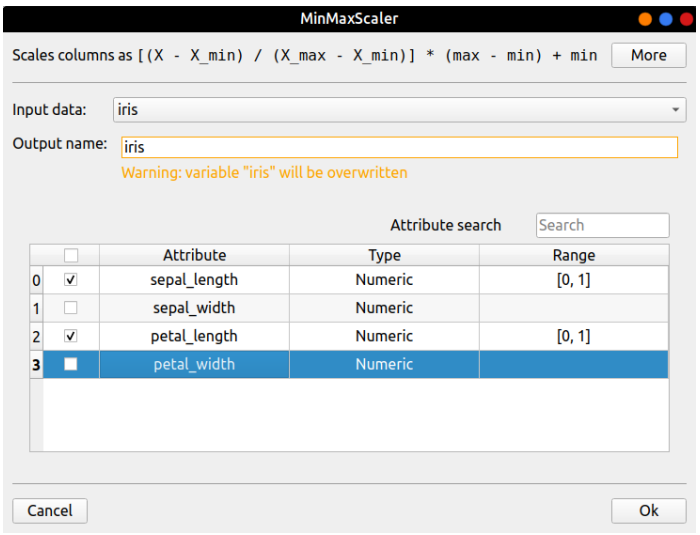


Figure 2.9: The editor widget for min-max scaling. The editor is also warning the user that the output name already exists in the workbench, and thus the output of the operation will overwrite the current dataset.

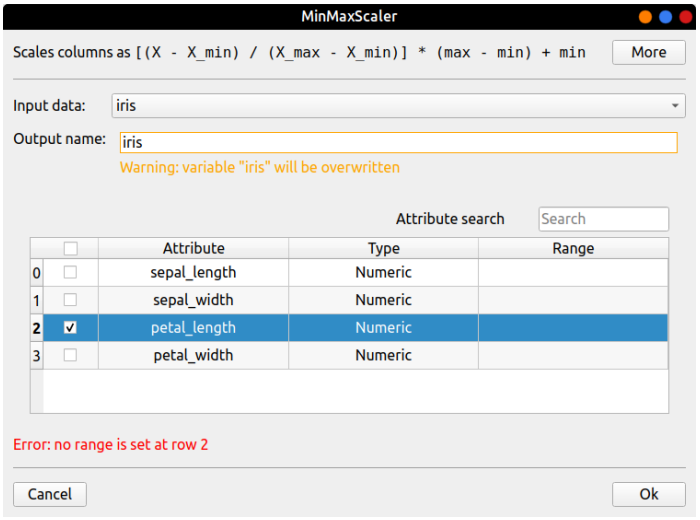


Figure 2.10: The min-max scaler editor widget displaying a validation error

editor widget. Fig. 2.9 shows the editor used with the min-max scaling operation applied to two numeric attributes of the *iris* dataset. In this example it requires the user to specify the scaling range for the selected attributes.
When some configuration parameters are needed the software should also ensure that the combination of options is always correct. Typically some transformations are not applicable to every data type: for instance, discretization can only be applied to continuous-valued attributes, which in DataMole are only *numeric* and *datetime* columns. One-hot encoding can reasonably be applied only to *string* or *nominal* columns. To assist the user and to prevent him from selecting wrong types, whenever he is required to choose the set of columns to be transformed during options configuration, the widget editor only shows the subset of columns with a supported data type. If more than one type is supported, the operation must also take care to handle processing of each type correctly.
Sometimes different data types require considerably different treatment, even for a conceptually similar transformation. In this situation the two data types are handled separately in two distinct operations. This is the case for *numeric* and *datetime* discretization: the latter requires a specialised editor widget to handle dates and time imputing, as well as a very different treatment during execution, compared with its numeric equivalent. In this situations specialising operations is convenient to simplify the editors, improving error handling and consequently ensures a better user experience.
In order to prevent runtime errors as much as possible, every option set by the user is *validated*. Any detected configuration error is notified by displaying error messages directly on the editor widget. The user needs to correct the errors to be able to set options correctly. Fig. 2.10 shows a validation error message on the editor widget for the min-max scaling operation. In this case the user tried to confirm the options without setting the scaling range for the selected attribute.
Sometimes its not possible to prevent *runtime errors*. Besides, there are situations in which runtime errors are actually intended to happen. As an example, some operations for type conversion ask the user how they should behave when a value which can not be converted is encountered: the user may choose to be informed of this to manually solve it, or he may tell the operation to set every unsupported value to NaN. If the user chooses the former, the operation may stop with an error and a message will be shown telling him that conversion was not possible.
Additionally, even if DataMole tries to prevent errors, there is the possibility that some combination of parameters and inputs could break the operation. For example, some operations fail if they are applied to a column with only missing values. This is the standard behaviour of many Python packages and generally corner-cases like this are not prevented from happening.
As a consequence, every runtime error or warning is reported to the user with a pop-up notification.

2.3.5 Logging transformations

Every operation, even the ones applied from the *Attribute panel* are logged. Log files are placed in the `logs` folder within the main program directory. These logs are not meant for debug purposes, but merely to keep a trace of the operations that were applied to any dataset during a program session.
Every time a pipeline is run from the *Flow panel*, its execution is logged in a new file created in the `graph` subfolder. For every applied operation the log contains its configuration (i.e. which options were set), eventual execution information and a *diff* of what changed in the dataset shape after the operation completed, including new and removed columns and changed types.

2.3.6 Exporting and interoperability

DataMole includes many features often required in data analysis but many others are missing or may be required for very specific use cases. Thus, it is essential to give the opportunity to use this software together with other Python libraries. To guarantee interoperability of this tool with external packages, every imported dataset can be exported as a Pandas dataframe using the `pickle` binary format. Almost every Python object can be exported in a `pickle` file, hence this feature provides a simple way to use processed datasets with arbitrary Python scripts.
As already mentioned, the pipeline can also be exported in this format. However pipelines can only be interpreted by DataMole, so this functionality is meant mainly to save pipelines for later use or to share pipeline objects between DataMole users.

Chapter 3

Development

This chapter contains an overview of the software architecture, describes the program external dependencies and explains how tests were conducted.

3.1 Technology

This section describes the technologies that were selected to realise DataMole and all the tools that supported its development.
DataMole was developed in Python in order to maximise interoperability with other packages and to take advantage of the existing data analysis libraries. The main dependencies are listed below:

- **Pandas 1.0.5**: Pandas dataframe is used to manage datasets and its rich API is used in the program to apply transformations and manipulate the data;
- **Scikit-learn 0.23.1**: this library is used to apply some transformations, because of its excellent compatibility with Pandas;
- **Networkx 2.4**: a library for graph management and analysis used to create and manage the computational graph;
- **Numpy 1.19.1**: a library for scientific computing and number crunching;
- **PySide2 5.15.0**: contains the Python bindings of the Qt Framework version 5.15.0 and is used to create the graphic interface;
- **Prettytable 0.7.2**: a library for printing formatted ASCII tables, used with logging;
- **PyTest 5.4.3**: a framework for writing tests in Python;
- **Sphinx 3.2.1**: used to automatically generate the documentation for the project.

We decided to use Pandas as the data management library because of its mature development state and richness of features. Pandas does not scale to big datasets and

does not support out-of-core dataset processing. In our case scalability was not a concern, since DataMole was required to work with relatively small datasets. The next section describes some alternatives that were discarded in the first place, but may be worth reconsidering for future extensions.
Many frameworks can be used to create a desktop application with Python, like Tkinter, WxPython and Kivy. Qt is a C++ framework often used to create graphical interfaces. It can be used with Python by installing the PyQt or the PySide package. PyQt is maintained by a third party company [23], while PySide2 is developed by the same company developing Qt [19]. Additionally PySide2 is released under GNU LGPL license, while PyQt is licensed with the more constrained GPL v3.
In comparison to other frameworks, Qt is more complete and it is often suggested for building professional desktop applications. Additionally I already had experience with this framework in the past, so I was already accustomed to some of its usage paradigms.

3.1.1 Dataset management libraries

Pandas dataframes are used in DataMole to manage datasets, but other libraries offering similar data structures that were taken in consideration and are listed here.

- **Dask**: a project started in 2015 with the goal of creating a distributed computing library with big data support. It offers a dataframe API very similar to Pandas, with the difference of being able to manage huge out-of-memory datasets;
- **Datatable**: a Python porting of the popular `data.table` package for R, developed by the same authors. Only a beta version is currently released, and many features of the R version are missing. Similarly to the R package it can deal with out-of-memory datasets;
- **PySpark**: a big data computing framework written in Scala which supports data stream, map-reduce operations on distributed file systems;
- **Turicreate**: a project currently maintained by Apple with the goal of simplifying the development of custom machine learning models. It supports big data computations and provides the SFrame container, a scalable dataframe similar to a Pandas dataframe;
- **Modin**: a project with the goal of scaling Pandas to big datasets. Internally is uses Dask or Ray to transparently process dataframes and to support out-of-core and parallel computations.

Both `Spark` and `Dask` are designed to work with big data and heavy computations: internally they use a scheduler and add complexity to support distributed file systems. This focus on big data often results in additional computational overhead, which is unjustified for the purpose of this project at its current state.
`Datatable` and `turicreate` are much simpler to use than the previous libraries, since they do not handle distributed computation. `Datatable` main disadvantage is its very

limited API with respect to Pandas and the missing support for Windows. `Turicreate` is more limited then Pandas, but still comes with a rich set of features that makes it probably suitable for this project. It has one limitation: it works only with 64-bit machines.
Finally `Modin` was discarded because it is still an experimental project.
Adding big data support is of course desirable, but comes with additional costs and overhead that should be taken into account. With respect to this project objectives, I found the additional complexity not worth it, especially for an initial release, and decided to work with Pandas. This library has also the advantage of being well integrated in the Python ecosystem, and its dataframes are supported by other packages like Scikit-learn. Naturally the other listed libraries may be reconsidered for future extension, if scalability becomes a concern.

3.1.2 Development environment

DataMole was developed using Python 3.8.0 under Ubuntu 18.04.5 LTS using JetBrains PyCharm IDE. I used the LaTeX editor TexStudio for documentation and PlantUML with Visual Studio Code to create UML diagrams.
The project and its documentation are versioned using Git and the repository with the code is hosted on GitHub.

3.2 Qt basics

This section briefly describes two important features of Qt that have been widely used to develop DataMole. The purpose of this section is not to provide a comprehensive description of the Qt Framework and its functionalities, but merely to clarify the meaning of some technical terms that will often be used throughout this chapter.

3.2.1 Signals and slots

Signals and slots are used for communication between objects and their combined usage represents the Qt approach to event-driven programming [17].
A *signal* is emitted when a particular event occurs. For example when a button is clicked, the `clicked` signal is emitted. A *slot* is a member function that is called in response to a particular signal. Qt widgets come with many predefined signals and slots, but they can be subclassed to add new customised slots handling specialised signals.

3.2.2 Model-view-delegate

The Qt Framework provides a set of classes that use a model/view architecture to manage the separation between data and the way it is presented to the user [16]. This is achieved with the combined usage of three components:

- *Model*: model classes inherit `QAbstractItemModel` that provides an interface for the other components in the architecture. The model communicates with the data: it can hold the data or it can be a proxy between the data container and the other components;
- *View*: view classes inherit `QAbstractItemView` and their instances obtain references to items of data from the model. With these, a view can retrieve items from the data source and display them to the user;
- *Delegate*: a delegate renders the items of data inside a view and, when items are edited, it communicates with the model to change its state. Every delegate class inherits `QAbstractItemDelegate`.

These components define the *model-view-delegate* pattern, which is extensively used to show information in DataMole.

3.3 Architecture overview

This section describes the architectural choices made while designing the DataMole software architecture. UML class diagrams are presented alongside packages description to model the relation between different components. Moreover, several sequence diagrams are shown in §3.3.4.5 to show the interaction between GUI components that is required to carry out some operations.
The extension mechanisms, some implementation details, as well as a detailed description of various classes and their methods, are included in the developer manual in Appendix A.

3.3.1 Description of the main packages

DataMole is a Python package, composed of many sub-packages and modules. In Python notation a *module* is a single file containing definitions and statements, while a *package* is a collection of modules (e.g. a directory containing Python files and possibly other packages) [10]. The complete directory structure can be seen in Appendix A.
The main DataMole package, named `dataMole`, is organised in 4 sub-packages:

- *data*: defines the container classes for dataframe objects and everything required for their management. Every Pandas dataframe is wrapped inside `Frame` object. Every dataframe has a `Shape`, which is a description of the columns names and their types. Finally a hierarchy of classes was defined to model the supported data types;
- *flow*: defines the data structure used to contain the flowgraph (`OperationDag`) and the handler to execute it in separate threads (`OperationHandler`);
- *gui*: contains all the widgets and GUI components used within the software. It is further composed of 4 sub-packages:

  - *charts*: contains the widgets used to create and visualise all the charts;
  - *editor*: defines the abstract class `AbsOperationEditor`, the super-type of all editor widgets used for operation configuration. It also contains the definition of the editor factory `OptionsEditorFactory`, defined as a *singleton*, and many utilities to configure the operation and show the operation documentation;
  - *graph*: collection of components which realise the user interface for the *Flow panel*. It relies on the *Qt Graphics View Framework* and part of the implementation was taken from an existing work available at [5];
  - *widgets*: contains the definition of several widgets used in the program.

- *operation*: defines the common super-type of all operations and its subclasses, which model every data transformations;

- *flogging*: the DataMole logging module. Also defines the interface to be implemented in every operation whose execution must be logged to file (`Loggable`).

3.3.2 Model/view classes

Many classes that take part in *model-view-delegate* pattern are defined inside module `gui.mainmodels`. Fig. 3.1 shows the UML class diagram for some of its classes, along with their relationship to the `data` package.
Qt provides specialised models and views for displaying data in many different ways: within DataMole data is often shown in tables and lists. In order to take advantage of the Qt model-view system, model classes are required to subclass `QAbstractItemModel` and implement the relevant methods.
`FrameModel` handles the visualisation of the dataframe inside a `QTableView` (which is a view for tabular data).
`AttributeTableModel` works as a proxy model, keeping a reference to the `FrameModel` and showing only column names and types, hence is used to show the *shape* of a dataset, which is required, for instance, in the *Attribute panel*.
`AttributeProxyModel` further refines the data shown in an `AttributeTableModel` by adding filtering capabilities, like attribute search by name or regular expression and type filtering. This last feature is exploited in the operation editors that only allow the user to select the subset of columns with supported types.
When a dataset with thousands of columns or rows is visualised inside a table, the process of filling every table cell with the data from the model can take time, and the GUI would be unresponsive while this happens. Proxy class `IncrementalRenderFrameModel` solves this problem: it shows the exact same data as the `FrameModel` it proxies, but implements additional methods, specifically `fetchMore` and `canFetchMore`, to make sure the table is filled on demand, only when it is scrolled. When this happens it loads the next batch of 50 columns or 400 rows depending on the scrolling direction.
`SignalTableView` defines the view used to show tabular data in the program. It inherits `QTableView` capabilities and defines a customised *signal* used to correctly handle rows

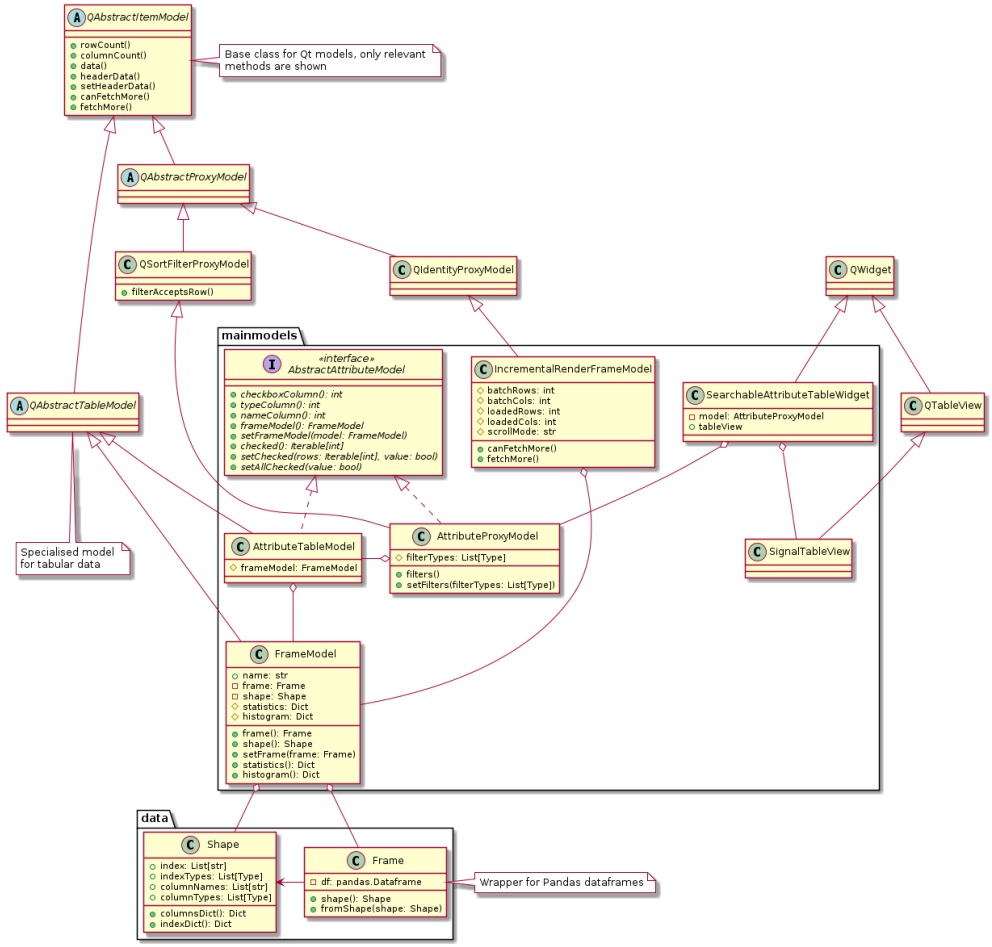


Figure 3.1: Class diagram of the `mainmodels` module

selection and deselection. This class is used inside the `SearchableAttributeTableWidget`, a reusable widget that shows data inside a table complete with a search bar above it. Finally the module also defines a custom delegate class and a custom header for the table (not shown in Fig. 3.1), to be set in views that need to show a checkbox in one of their columns.

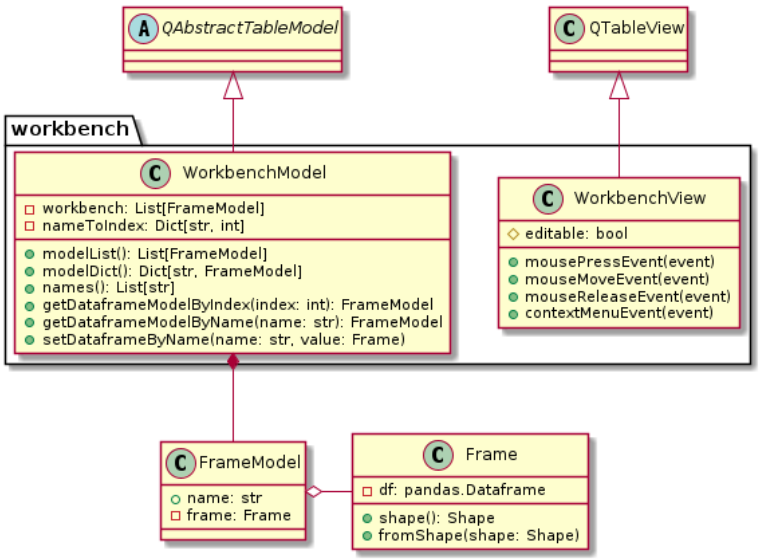


Figure 3.2: Class diagram of the `workbench` module

3.3.2.1 The workbench

As explained in the previous section every loaded dataframe is contained in a `Frame` object. These objects are wrapped in a `FrameModel` for visualisation inside Qt views. All these datasets are kept in the `WorkbenchModel`, the centralised container for dataframes (diagram in Fig. 3.2). It provides methods to access frame models by row number (index) or name, that makes it usable as a list-like or dictionary-like container. Its `setDataframeByName` method is called whenever a new dataset, which was loaded or created by applying operations, must be added to the workbench. The workbench is shown using its specialised view class `WorkbenchView`, which overrides some methods to allow item reordering, selection and the display of a menu on right click.

3.3.3 Representation of a dataset transformation

When defining a pipeline, every operation takes one or more datasets in input and produces a result that must be the input of the next transformation. Sometimes transformations must be parametrised with a variable number of arguments that represent the configuration of the operation. Additionally every transformation should be logged, in order to keep a trace of how a dataset was changed and, depending on the operation, it might be possible to undo it.
With these requirements it seemed convenient to implement a *command pattern*: every *data transformation*, also called *operation* in this document, is encapsulated in a object of type `Operation`, whose contract is described in the next section.

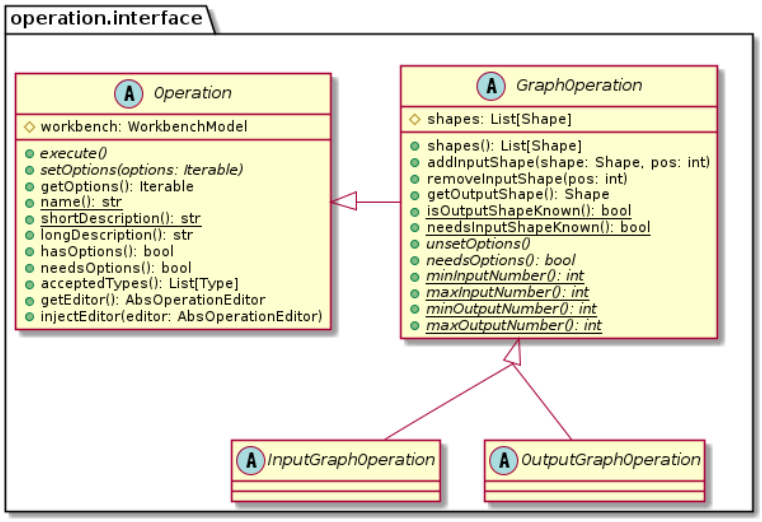


Figure 3.3: Hierarchy of Operation abstract classes

3.3.3.1 The *Operation* abstract class

Every operation defined in DataMole is a concrete subclass of `Operation`. Fig. 3.3 shows the `Operation` abstract class and its subclasses.
An operation keeps a reference to the workbench, which provides access to any loaded dataset. Every operation has a name, comes with a short and optionally a long description to be shown in the *editor widget*. The long description is shown on user request on a separate help panel, and contains a detailed description of how the operation works and how it should be configured.
The `execute` method is run to apply the operation to its input arguments, which are passed as parameters.
The `getOptions/setOptions` methods are reimplemented in every operation to respectively get and set its arguments. If required, `setOptions` can also validate the new options before setting them. If they are not correct it raises an exception of type `OptionValidationError` which is handled by the controller. This mechanism is further described in §3.3.4.5.5.
Methods `hasOptions` and `needsOptions` return a boolean value depending on whether the operation is configured or needs an option editor widget. If the latter is true, the operation also needs `getEditor` to be overridden to return the widget that the operation will use to be configured. In order to avoid the definition of a new widget for every operation, an *editor factory* was provided and can be used to quickly create editors with standard option fields (checkboxes, combo boxes, line edits, etc.).
While `getEditor` only returns a new editor widget, the `injectEditor` method is used to configure the editor.
Finally the `acceptedTypes` method defines which types are accepted by the operation.

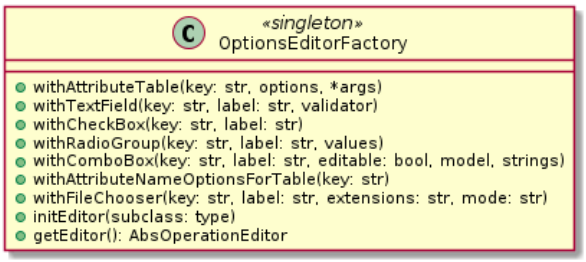


Figure 3.4: Widget factory class specification

3.3.3.2 The editor widget factory

Most operations require options that must be supplied by the user. For example the `KBinsDiscretizer` operation requires the user to select the columns to be discretized, specify the number of bins for every column and select the strategy to be used. To configure an operation, the widget returned by method `getEditor` is used. For most operations this widget is composed of a table with some editable columns and some radio buttons or checkboxes. To avoid code duplication and to speed up the process of creating widgets, a factory class was defined. Fig. 3.4 gives an overview of the class methods.
Factory methods allow appending widgets to the editor layout stacking them vertically, thus the order of invocation determines their ordering. The factory is a *singleton* object that must be re-initialised every time it is used by invoking the `initEditor` method. This method also accepts an optional `subclass` parameter with the type of the editor to be created. This is particularly useful when an editor needs signals or slots to be part of its class definition. To give an example, Fig. 3.5 shows the editor for the *Fill NaN* operation, created by calling the following factory methods:

- `withAttributeTable`: this method creates widget *(A)*, which is a table showing every attribute name and type, with a configurable number of extra editable columns used to receive options for every attribute. In this example the table requires the values to be filled if the option to fill by value is selected (column *Fill value*);

- `withRadioGroup`: adds a set of exclusive radio buttons and a descriptive title *(B)*: in the example it is used to select the strategy for filling missing values.

Further details about the factory usage and methods configuration are reported in appendix, in §A.2.4.2.
Of course not every widget can be created with the factory: complex editors with particular requirements must be defined manually. For instance, this is the case for the `Join` and `ExtractSeries` operations.

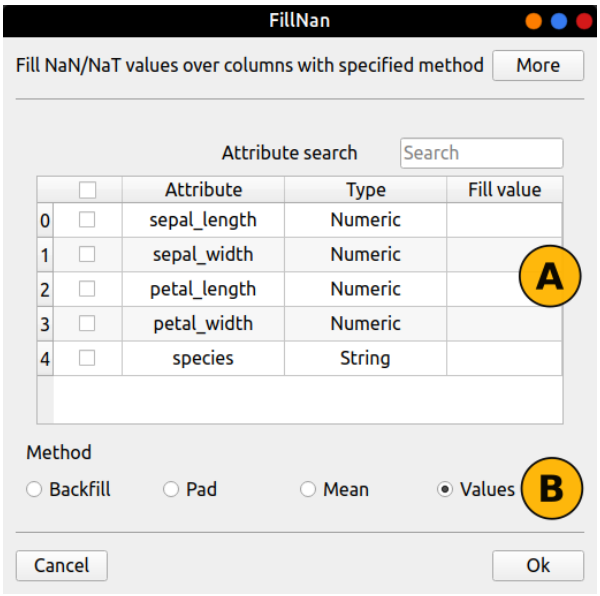


Figure 3.5: Example of an *option editor* whose body widgets (*A* and *B*) are created with the widget factory

3.3.4 The computational graph

As explained in §3.3.1, the package `gui.graph` defines the components of the graphical user interface which allow the user to interactively create a computational graph of operations: new nodes (i.e. operations) can be added, moved, deleted and connections can be created between existing nodes to create chains of operations. Hence, this interactive flowchart can be seen as a *pipeline* of data transformations, internally modelled as a *directed acyclic graph*. The class diagrams of the `gui.graph` and `flow` packages are depicted in Fig. 3.6.

3.3.4.1 Pipeline laziness

Whenever a new operation is added, it is not executed immediately: the pipeline is *lazy*, meaning that it is evaluated completely only when the user requests it. Thus every node with an ancestor does not know its input until the whole pipeline is executed. However most operations require to be configured with additional arguments through theirs editor widgets, and often this arguments depends on the operation input (for example the user may be asked to select the columns to transform), so in order to do this they *need* some information about their inputs. Additionally, every time the operation is configured, its options are validated, process that often depends on the column and index types. To solve these problems every operation needs to be able to describe some properties of its output before execution, provided a description of its inputs and the required options. Specifically, it should describe the *shape* of its output. The *shape* of a `Frame` object is encapsulated inside the type `Shape`: it includes information about the name and type of every column, including the ones used as

dataframe index. How the dataset shape is propagated through the graph is described in the following sections.

3.3.4.2 The *GraphOperation* abstract class

The additional requirements of the operations that need to execute inside the pipeline motivated the definition of the `GraphOperation` class, that specialises `Operation` adding methods only relevant in the pipeline context. Every graph operation needs to define how many inputs it needs and how many output connections it supports. This information is conveyed by the `minInputNumber`, `maxInputNumber` and their respective counterparts for the output number.
Moreover every graph operation reimplements method `getOutputShape`, which returns the `Shape` object describing the names and types of every column in the dataset after the application of the operation. Because this information needs to be propagated through the graph, operations also provide the `addInputShape` and `removeInputShape` methods, to manage the shapes object when some configuration changes or connections are removed.
However, there are situations when it might not be possible to predict the output shape before the operation is run. If this is the case, method `isOutputShapeKnown` can be redefined to return *False*. This is the case for the `RemoveBijections` operation, which automatically removes every column which is a bijection of the others. Conversely some operations might not require knowing in advance the shape of the input coming from their incoming connections, in which case `needsInputShapeKnown` should be redefined to return *False*. For example `RemoveNanRows`, the operation used to remove every row with a certain ratio of missing values, does not need to know the input dataset shape because its output shape does not depend on it: since the operation just remove rows, the output shape does not change at all.

3.3.4.3 The graph data structure

The operations added to a pipeline are stored inside a `DiGraph`, a data structure that models a direct acyclic graph, provided by `networkx`, a very popular network analysis library. This data structure is wrapped into an `OperationDag` object, that contains some helper functions to manage the graph of operations.
Before choosing to implement a simple computational graph using `networkx`, these Python libraries specific for creation and management of computational graphs were also considered:

- **Luigi**: a package that simplifies the creation of pipelines of batch jobs;
- **GraphKit**: a library used to manage and run graph of computations (DAGs);
- **Dask**: this library offers many tools for scalable computations and its schedulers support the execution of customised task graphs.

Eventually we decided to avoid using such libraries, since computations in DataMole are not particularly heavy and thus would not have benefited from the advanced

features provided by these packages. Additionally integrating them in a graphical interface would require much work and time, because their API is not made for this use case, but rather for usage in Python scripts.

3.3.4.4 The GUI for the graph

GUI components used to manage the graph are defined in the `gui.graph` sub-package. This package makes use of the *Qt Graphics View Framework*, the Qt module for efficient rendering and management of graphic items [15]. Many modules of this package were adapted from [5], a project that used Python and Qt to build a generic graph manager, and is licensed under GNU GPL.

3.3.4.4.1 Main graphic components

The *Qt Graphics View Framework* relies on three main classes:

- `QGraphicsItem`: represents a graphic item that can be shown within a graphics scene;
- `QGraphicsScene`: the scene manages all the graphic items and provides functionality to efficiently determine items location, and control zooming and selection;
- `QGraphicsView`: visualises the content of a `QGraphicsScene`.

Fig. 3.6 shows the classes defined in the `gui.graph` package. All graphical items are contained in the `GraphScene` a subclass of `QGraphicsScene` that additionally handles drag-and-drop actions and mouse events. Graph *nodes* and *edges* are respectively drawn using the `Node` and `Edge` classes, both subclasses of `QGraphicsItem`. Every `Node` has a unique id that matches the id of the operation it represents. The `NodeSlot` item is used to draw the circular sockets that represent the operation inputs and outputs and are used to create connections. Finally `RubberBand` is used to handle mouse selection. A *controller*, modelled by class `GraphController`, is used to ensure that the graphic interface is kept synchronized with the underlying data structure containing the graph. Every graph node is an object of type `OperationNode` which wraps a `GraphOperation` and adds some helper methods used to manage pipeline nodes.

3.3.4.5 Pipeline management workflow

The `GraphController` interprets the user actions on the view and keeps updated the graph data structure inside the `OperationDag` object. This section describes how the controller and the other classes interact with each other to manipulate the pipeline.

3.3.4.5.1 Node creation

Nodes are placed on the graphic scene with drag-and-drop, by selecting operations from a list and dropping them on the graphic view. When a new operation is dropped, method `dropEvent` of `GraphScene` is called and the *scene* invokes method `getDropData`

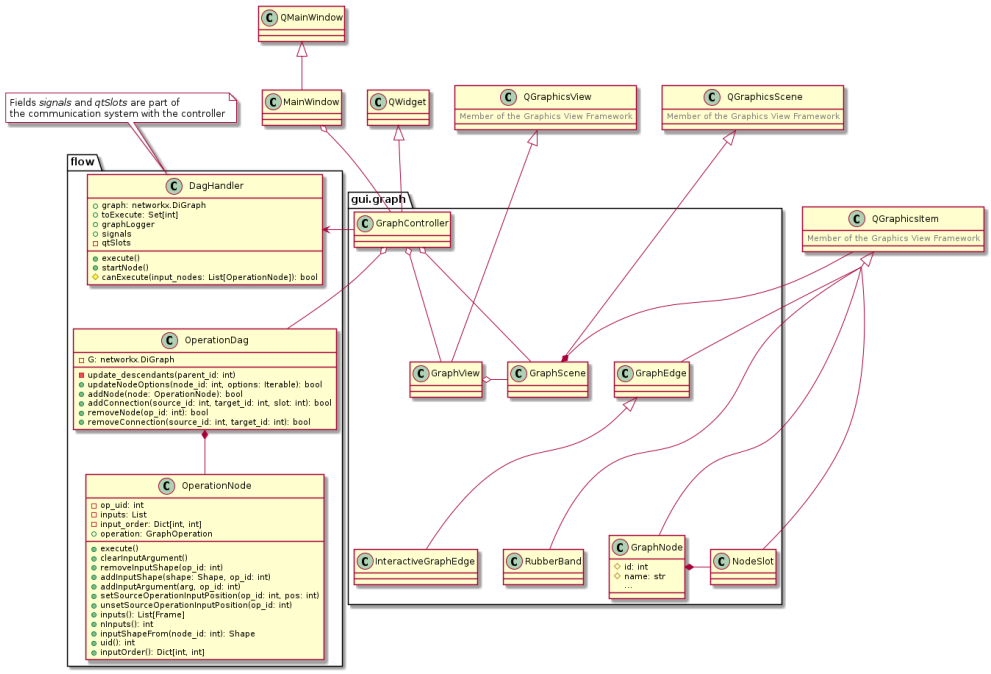


Figure 3.6: Class diagram of the `gui.graph` and `flow` packages

to retrieve the type of the operation that was dropped. It then emits a signal to ask the *controller* to manage the creation of the new operation. The controller instantiates the new `Operation` and `OperationNode` objects, updates the graph with the new operation and, if successful, it updates the graphic scene with the new node. This entire process is depicted with the sequence diagram in Fig. 3.7. On the other hand, if something goes wrong and the pipeline cannot be updated, the controller shows an error message and nothing is changed.

3.3.4.5.2 Edge creation

When the user starts to drag a new edge from a `Node`, it emits a signal and causes the `start_interactive_edge` method of the *scene* to be called. Its purpose is to display an interactive draggable edge that exits the source node and follows the mouse pointer until the user drops its head on the target node. When this happens `stop_interactive_edge` is called. This method looks for a free node slot in the target node. If a free slot is found the `addEdge` method of the *controller* is called to update both the acyclic graph and the graphic view accordingly. The corresponding sequence diagram is shown in Fig. 3.8.
If the edge cannot be added, for example because the source operation does not provide an input shape which is needed by the target operation, an error message is shown to the user. In either case the temporary `InteractiveEdge` object created to display the draggable edge is eventually deleted.

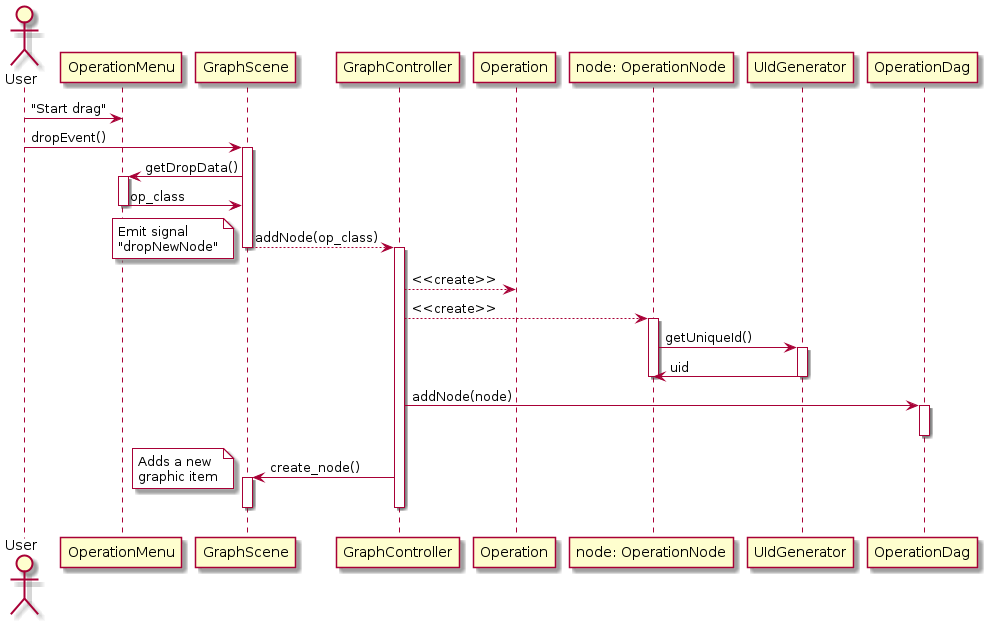


Figure 3.7: Sequence diagram describing the creation of a new pipeline node

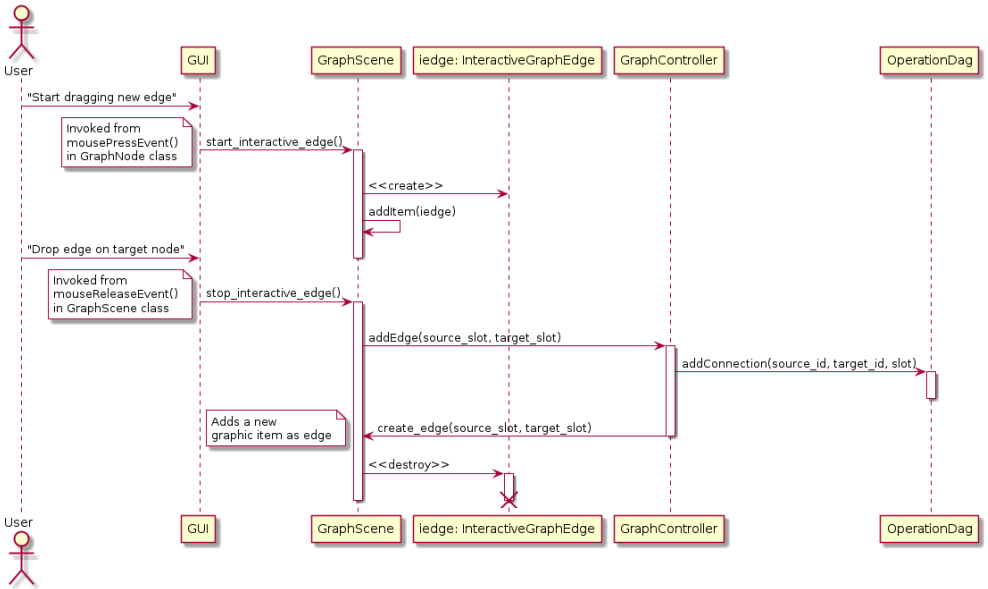


Figure 3.8: Sequence diagram for connecting two existing operations

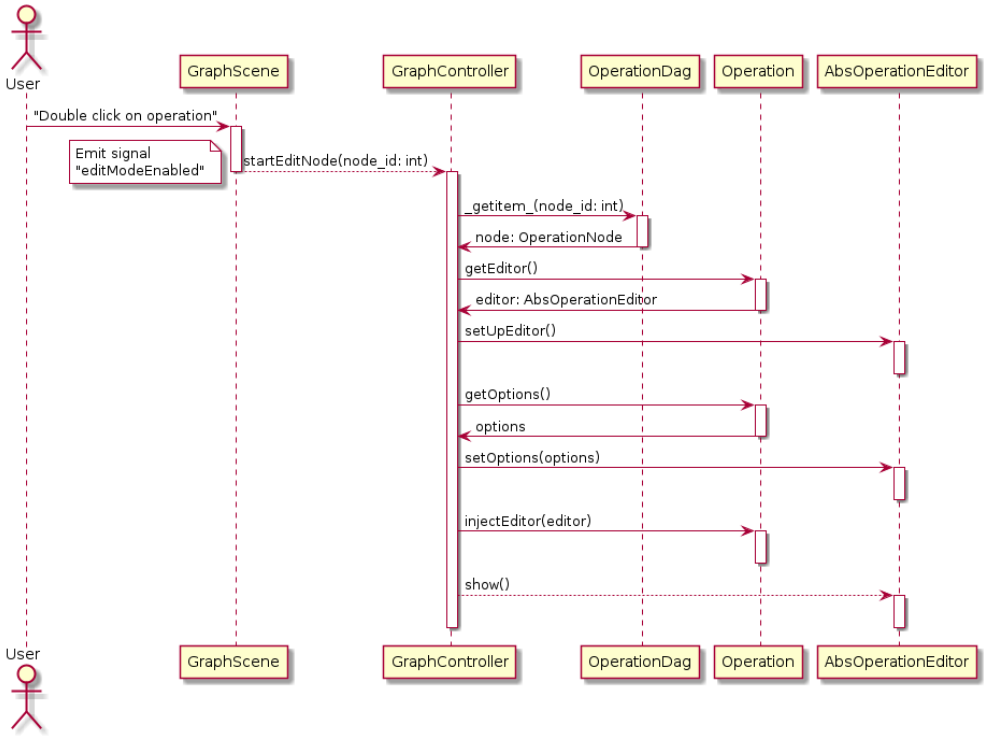


Figure 3.9: Sequence diagram for operation configuration

3.3.4.5.3 Operation configuration

Double clicks on existing operations cause the editor widget to be shown. Fig. 3.9 describes how the widget is created and configured. First the *controller* retrieves the `Operation` object corresponding to the clicked item using its unique id. Afterwards the editor is created using the `getEditor` method and the existing options are retrieved from the operation and set into the editor widget. Finally the editor is configured with the `injectEditor` invocation and shown to the user.

3.3.4.5.4 Option confirmation

Once the user sets new options in the configuration widget and confirms them, the `accept` signal is emitted by the editor. This signal is handled by the `GraphController` that retrieves the options from the editor and updates the corresponding operation. Finally the `update_descendants` routine is run to propagate the new shape through the graph. The whole process is shown in Fig. 3.10.

3.3.4.5.5 Option confirmation with validation errors

The `setOptions` method defined in every operation can raise an exception of type `OptionValidationError` if one or more options are not correct and need to checked by the user. This exception is parametrised with a list of tuples, where each one represents

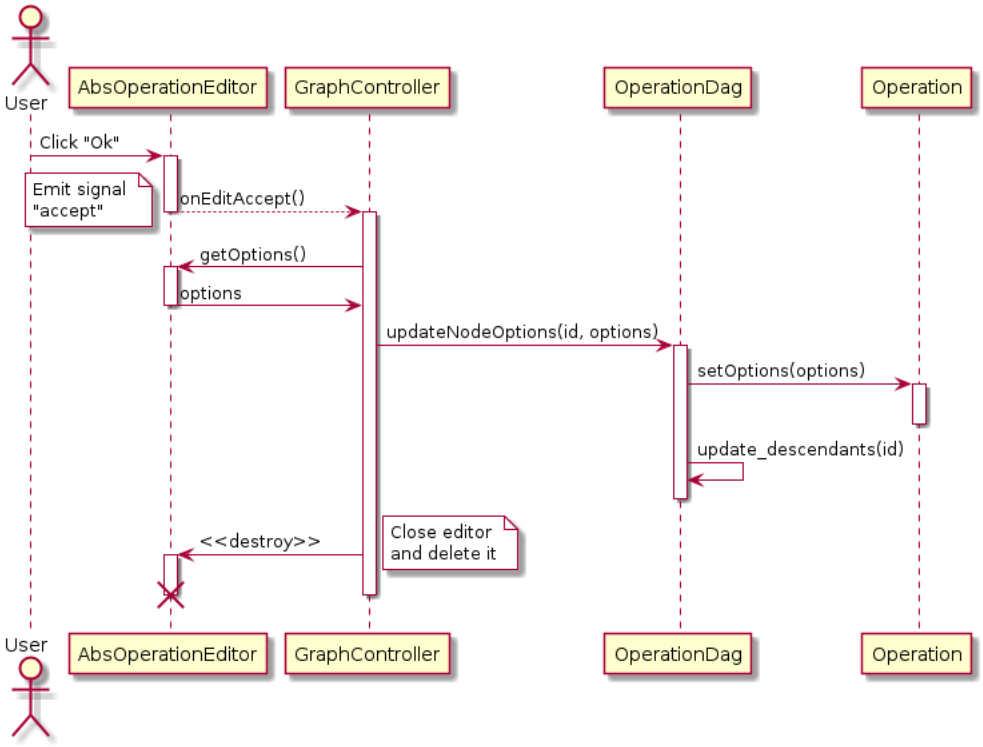


Figure 3.10: Sequence of operations after options confirmation assuming no validation errors occur

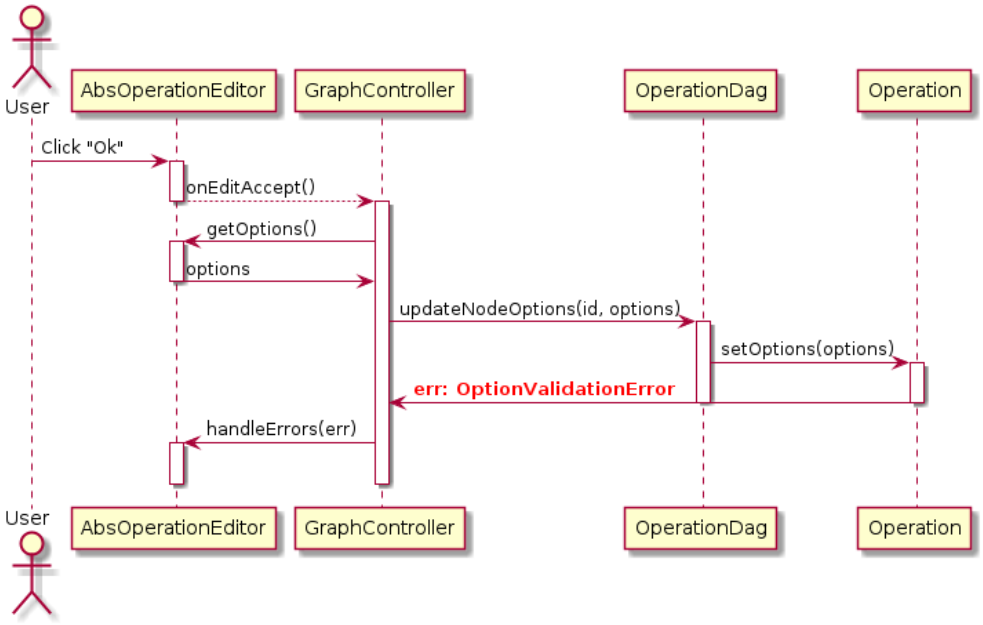


Figure 3.11: Sequence of operations after options validation exception

an error, with a key and an error message for the user. After the user confirms the options in the configuration widget, eventual validation errors are notified inside the editor window. The process leading to this outcome is outlined in Fig. 3.11.
Error messages are normally shown on the bottom of the editor widget. If a more advanced error handling is required, it is possible to change this behaviour by defining custom error handling functions. This is detailed in §A.2.4 of the appendix.

3.3.4.6 Executing the pipeline

Execution of the pipeline is handled by the `DagHandler` class. This handler is instantiated by the controller to direct the execution of the whole pipeline. First it checks for runnable operations: an operation is considered *runnable* if all the required options and its input dataframes are set. Starting from the input nodes, runnable operations are scheduled for execution on a separate thread. When an operation completes, its output is set as the input of all its successors and all runnable operations are started. Additionally the handler communicates with the controller emitting a specific signal when the status of an active operation changes (i.e. if it is running, has completed with error, etc.). This allows the controller to update the status of each node in the graphic view.

3.3.4.6.1 Multithreaded execution

In a Qt application, all QWidgets run in the *main thread*, also called the *GUI thread* [22]. If a time-expensive computation is run on this same thread the GUI will freeze and stop responding until the operation completes. To avoid this problem every operation to be executed is wrapped inside a `QRunnable` object and is added to a `QThreadPool` for execution. The combined usage of `QRunnable` and `QThreadPool` represents a simple multithreading pattern in Qt, with the advantage that the `QThreadPool` takes care of thread management, including their creation and destruction: it creates a predefined number of threads equal to the number of cores of the machine and reuses them whenever a new operation is added to the pool. Considering that thread creation and destruction can be costly, this pattern provides an higher-level alternative to thread management with the `QThread` class, which would instead create a new thread every time, without reusing them [20].
The helper class `Worker` was defined in the `threads` module, and is shown in Fig. 3.12. `Worker` objects wrap *executable objects*, which are arbitrary Python objects that define an `execute` method (like `Operation` instances). The worker is provided the list of arguments that should be passed to the `execute` method, if any, and an `identifier` to be used when a signal is emitted. A `WorkerSignal` class inheriting `QObject` is defined because `QRunnable` does not inherit `QObject` and thus can not emit signals. Through this class, workers emit three signals, depending on their state:

- *error*: this signal is emitted then the worker was running the operation but a runtime error occurred. The signal arguments are the operation identifier and the error information;

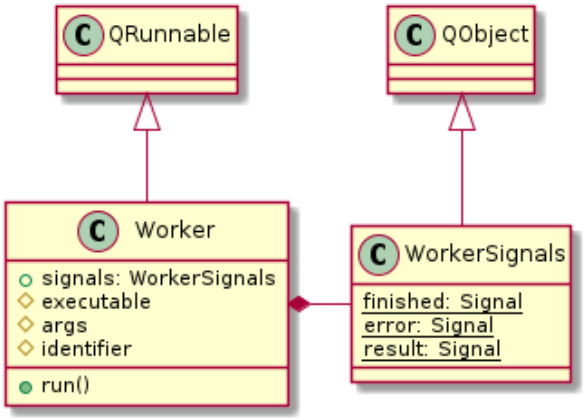


Figure 3.12: Class diagram of the `threads` module

- *result*: emitted then the operation completes successfully. It carries the operation result (i.e. the return value of the `execute` method) and the operation identifier;

- *finished*: signal emitted immediately before the `run` method of the worker returns, and carries the identifier.

These signals provide a way to monitor the status of every operation and are used by the `GraphController` to update the graphic view.

3.3.4.6.2 Considerations and alternatives

The `QRunnable` and `QThreadPool` pattern is a very simple multithreading pattern, and indeed was chosen for its simplicity. By comparison, the `QThread` class provides much more flexibility, at the cost of explicitly managing threads. The main drawback of the selected approach is the lack of support for stopping a running operation. This comes from the fact that `QThreadPool` does not expose the underlying threads and thus stopping them becomes impossible.
On the other hand, an equivalent approach using `QThread` may involve defining a customised thread pool that keeps track of the running threads, in order to terminate their execution when required. Nonetheless, caution is required when stopping a running thread as it may leave data in an inconsistent state.
Using the Qt Concurrent module would be a simpler alternative: this module, part of the Qt Framework, provides high-level functions to deal with some common parallel computation patterns [13]. Unfortunately this module is not included in the Python bindings for Qt. That is because the Qt Concurrent heavily relies on C++ templates and due to the Python dynamically-typed nature and the impossibility of generating C++ code at runtime there is not way to generate the Python bindings.

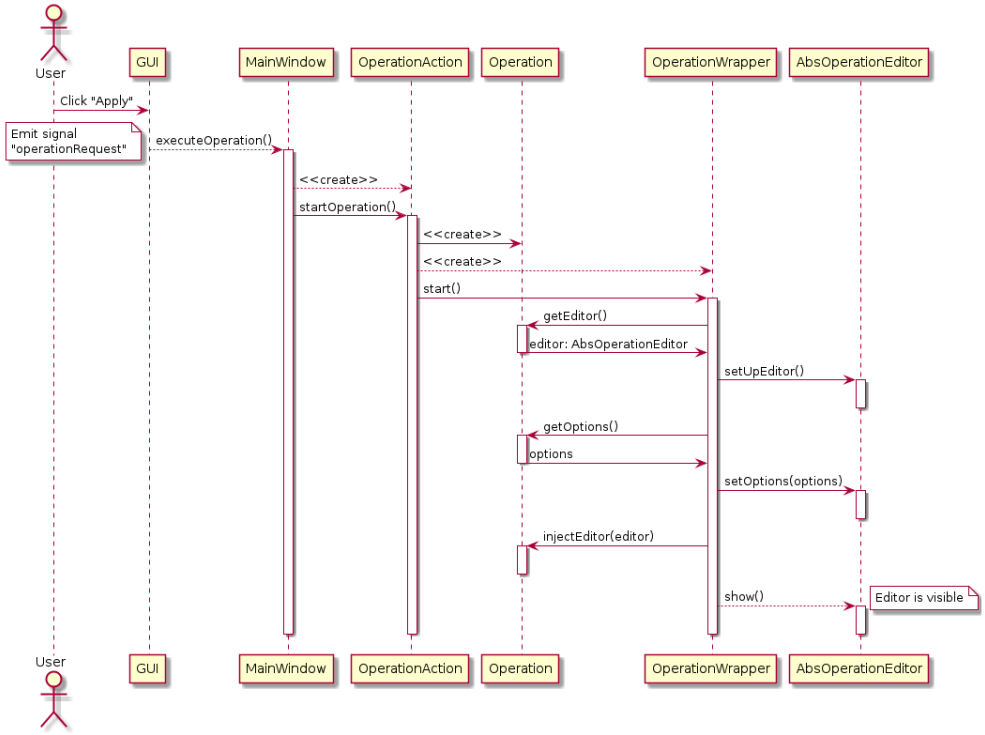


Figure 3.13: Sequence diagram of editor creation and display

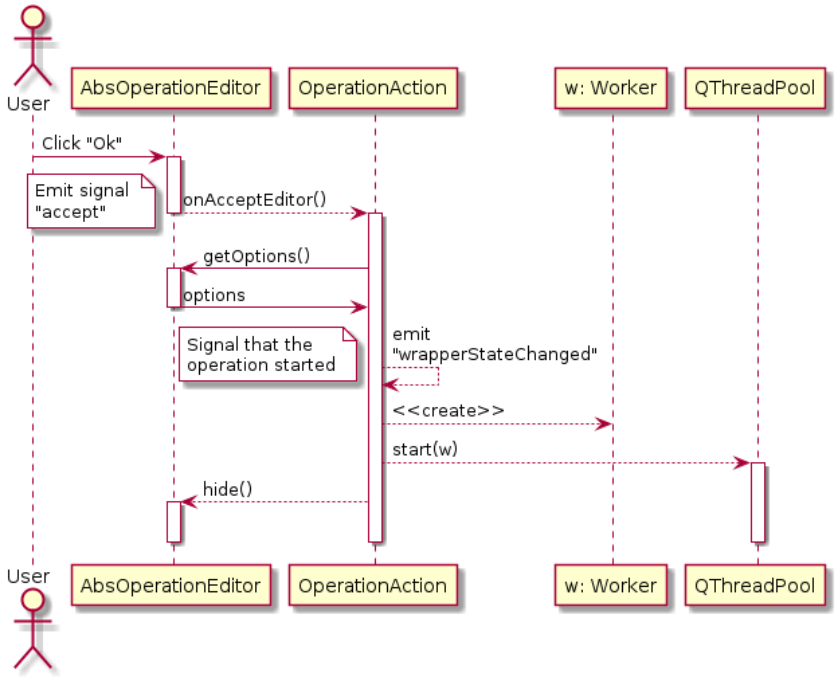


Figure 3.14: Sequence diagram of the option confirmation process

3.3.5 The *OperationAction* controller

A controller class named `OperationAction` was designed to manage the application of operations from the *Attribute panel*. This class inherits `QAction` which provides the support for launching operations from menu bars.
An `OperationAction` wraps an `Operation` and when *triggered*, either explicitly or implicitly (e.g. by clicking an action in a menu), it shows the option editor and waits for the user to set them. The whole process is described with a sequence diagram in Fig. 3.13. After confirmation, if validation succeeds the operation is executed: to do this a `Worker` object is created and added to the `QTreadPool` global instance. The sequence diagram for this is shown in Fig. 3.14. If option validation fails, errors are shown as already discussed in §3.3.4.5.5.
Because every `GraphOperation` is also an `Operation`, the `OperationAction` wrapper can also run graph operations as single commands, outside of the graph context. To execute such an operation as a single command there are two additional options that the user must provide: the input dataset and a name for the output. However editor widgets for graph operations do not provide a functionality to ask for these parameters. Consequently, whenever a graph operation is triggered, the `OperationAction` controller is also responsible of adding a combo box and an editable text box to the operation editor, before every other widget. The result can be seen in Fig. 2.9. The combo box allows to choose which dataset to operate on and the text box requires the name of the output dataset, that will be set on the workbench when the operation completes.

3.3.6 Charts visualisation

DataMole includes some visualisation features: it can show a frequency histogram to represent value distribution, a scatterplot matrix to visualise bivariate relations and a line chart for temporal series.
Implementing these features required some research about existing plotting libraries that could be embedded in this project, and they are reported in the next section.

3.3.6.1 Technological considerations

All charts in the program are drawn using QtCharts. This is the official module for creating graphs within Qt and is included in the PySide2 Python package, so no additional packages are needed. It builds upon the Graphics View framework and it is quite simple to use. On the other hand, it does not provide many advanced features, and its documentation is a bit lacking, with respect to the rest of the framework.
The following list contains a description of other packages that can be used to plot charts inside a Qt application:

- **PyQtGraph**: a scientific library based on Qt4 and `numpy`. It relies on the Qt GraphicsView framework and is released under MIT license;
- **QCustomPlot**: a Qt-based C++ widget for plotting and data visualisation

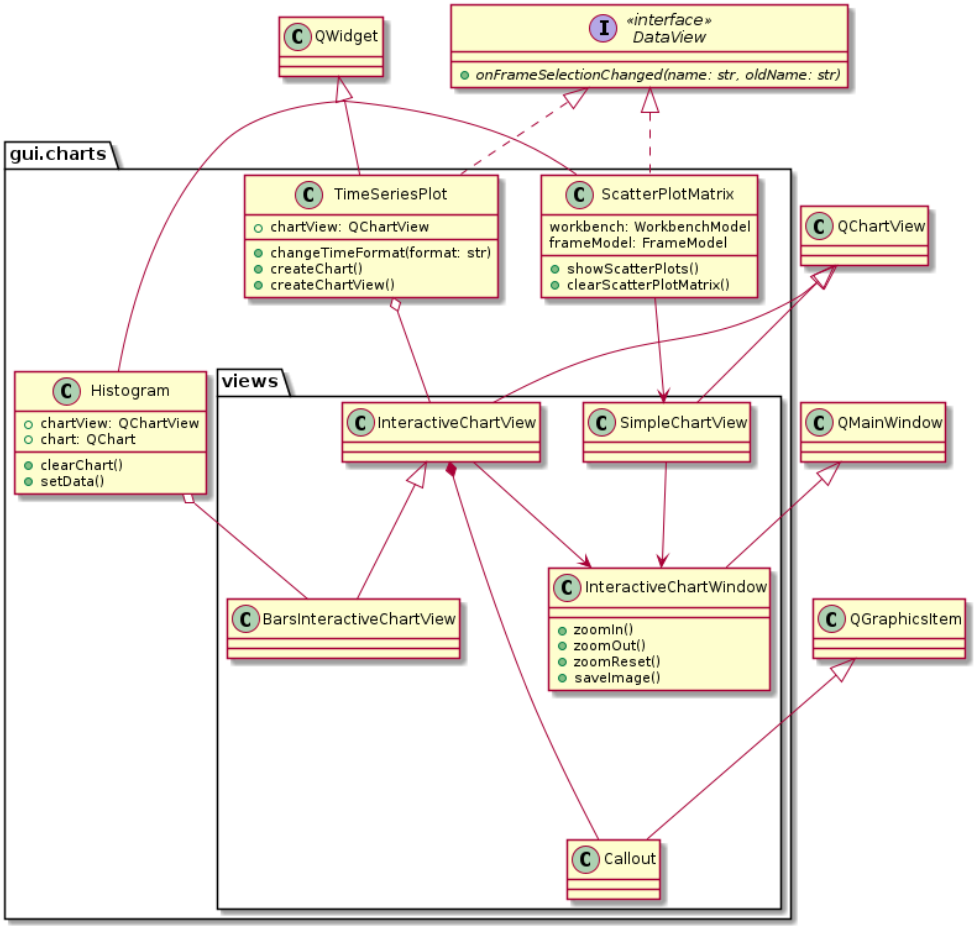


Figure 3.15: Class diagram of the `gui.charts` package

released under GPL v3 license

- **PyQwt**: another plotting library for Python based on PyQt.

PyQtGraph should be considered for future extension, since it is actively maintained and works with both PyQt and PySide2. It offers many advanced features that would require much work with the official QtCharts. On the other hand, QCustomPlot does not work with PySide2 and PyQwt is not maintained anymore, and only supports older versions of Qt.

3.3.6.2 The plotting package

Package `gui.charts` defines the widgets for creating and showing plots. Additionally customised chart views are defined inside the `views` module, and some related helper functions are in the `utils` module. Fig. 3.15 shows the package class members.
When a chart view is double clicked the chart is copied on a secondary window, of type

`InteractiveChartWindow`. This feature provides a way to move the charts around the screen as independent windows, and adds the ability to save its content as image. This feature relies on the ability to copy the chart data into a new one. Unfortunately this is not possible for the histogram chart, due to a limitation in the QtCharts module, and for this reason the `BarsInteractiveChartView` does not allow to open the chart on a new window. The `InteractiveChartView` also draws a pop-up item showing the data coordinates or label when one of its point is hovered with the cursor. This is implemented in the `Callout` class, which was inspired from the official Qt example available at [14].
`TimeSeriesPlot` and `ScatterPlotMatrix` are the widgets shown in the *View panel*, used to create the line chart for time series and the scatterplot matrix. Every widget that adds a visualisation feature to this panel must realise the `DataView` interface. It only requires the definition of one *slot*, which is called whenever the user clicks on a different dataframe in the workbench.

3.3.7 Logging

The `flogging` package provides all the logging functionalities for DataMole and internally uses the Python `logging` module. This package is used to create three different logs:

- **Application log**: contains all the application messages, like warnings, errors and debug messages. These logs are dumped inside the `logs/app` folder and are mainly useful for debugging;
- **Operation log**: logs the operations applied directly from the *Attribute panel*. Every log file is created inside the `logs/operations` folder and contains a summary of every operation run during a program session;
- **Graph log**: logs the execution of a pipeline. A different file is created every time a pipeline is executed inside the `logs/graph` folder.

Additionally the `logging` module also creates a root logger which logs everything passed to any active logger. This logger output is redirected inside the `logs/root` folder.

3.3.7.1 Logging operations

Every operation supports logging. Operations log their options configuration and every parameter set by the user.
Every operation that needs to be logged implements the `Loggable` interface. Diagram in Fig. 3.16 shows its two methods:

- `logOptions`: returns a formatted string with the configuration of the operation (typically user options) or other things that can be logged before the operation is run;

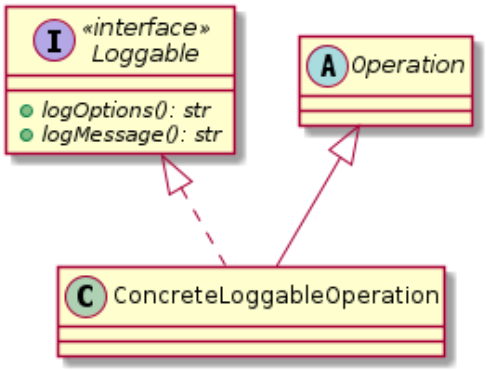


Figure 3.16: Class diagram for a sample operation that can be logged

- `logMessage`: returns a string with everything that should be logged after the operation completes, possibly including execution details.

Every time a pipeline is run the `OperationHandler` creates a new log file and logs the operations while they are executed. Operations launched from the *Attribute panel* are instead logged by the `OperationAction` controller. In either case, only operations realising the `Loggable` interface are logged.

3.4 Testing

Every operation defined in DataMole has been tested to ensure that input data are always treated correctly and produce a correct output. The `Pytest` package has been used to create test suites since it is very easy to use and require no boilerplate code. Every test suite is defined in a separate file inside the `tests` folder. Every operation is tested to ensure that every accepted data type with any possible combination of options is handled as expected and that exceptions are thrown when invalid options are set.
Module `mocks` contains the *mock class* definition for the `WorkbenchModel` and the `FrameModel` class. This is necessary because they are both subclasses of the Qt model classes, and thus they cannot be instantiated outside of a Qt Application, which is not initialised for testing purposes.
These unit-tests have been essential to define operations that correctly operate with Pandas and Scikit-learn. In addition, the `OperationDag` methods are tested to ensure that the graph data structure is updated correctly.
GUI components were not tested. To do this it may be worth using `pytest-qt` [12], a Pytest plug-in that allows to simulate user interaction to test widgets.

Chapter 4

Conclusions

At its current state of development, DataMole is a valid choice for exploration of new datasets. It embeds all the essential features described in §1.1: it can be used to preprocess multiple datasets, to draw charts, and allows to define, execute and save pipelines from within the graphical interface. Interoperability is achieved through the possibility to export Pandas dataframes with the `pickle` Python module. Finally it can be used to extract time series from longitudinal datasets, like ELSA and HRS, in order to easily visualise the progression of consecutive measurements in a line chart.
The proposed quality requirements were also met: the plug-in-oriented architecture ensures that new data transformations and widgets can be added with little effort. A developer and user manual have been produced and will be published inside the repository along with all the code, which is released under GNU GPL v3. Drafts of both manuals are also attached to this document in appendix. The developer manual is mainly focused on the description of the extension mechanisms embedded in DataMole, while the user manual describes the graphical user interface and explains how to use it.
However DataMole lacks many features with respect to related software like Weka and DataPreparator, hence further work is needed to take it to a comparable level.
The value proposition of this tool is centred around two core features:

- The possibility to build pipelines;
- Its plotting capabilities.

From a technical point of view these were the most challenging features to design. The first one required a research into packages created for this purpose and considerations about an appropriate software architecture to support it. Additionally the creation of the graphical interface for the *Flow* panel was quite complex and required time to understand how to use the Qt Graphics View Framework to build it, since it does not use the model/view paradigm which I already knew. As previously stated, the existing software found in [5], licensed under GNU GPL, was used as the groundwork for building all the pipeline graphic components.
The QtCharts module used to plot charts relies on the Graphics View Framework as

well. The main drawback of this module is the lack of support for some common chart functionalities, which required additional work to be implemented, and the scarce documentation.
Besides, using Qt from Python introduces some memory management problems that must be taken care of. Qt has its own garbage collection mechanism for every subclass of `QObject`, which ensures that whenever an object is destroyed, all its children are deleted too. Still, in order to avoid cluttering the main memory with widgets that are not used anymore, explicit deletion is used throughout the program. Since Qt objects live in a C++ runtime environment, this requires a synchronization between the C++ side and the Python side, because deleting a reference from just one side does not necessarily delete it from memory.
Qt for Python was still a good choice overall: it is a very powerful and complete framework, often considered one of the best for building desktop applications. On the other hand, Qt has a steep learning curve and thus requires some time to get familiar with. Looking ahead, the next major release of Qt, called Qt 6, which will be released progressively during the next years, promises many new features and a better integration with Python and its packages, including Pandas and Numpy [21]. As a consequence, future work may benefit from these improvements.

4.1 Packaging DataMole

Currently DataMole can be installed by cloning or downloading the GitHub repository and installing the required dependencies. Ideally it could be distributed as a standalone executable runnable on any machine, possibly using an installer. *PyInstaller* was tried in order to bundle DataMole and every dependency in a single executable file: unfortunately it did not work as expected, since some Qt modules were not included, thus creating problems during execution. Additionally PyInstaller bundles only work in the operating system where they were generated.
For now the simple installation from the repository suffices, but other packaging alternatives, like *fbs*, may be tried in the future.

4.2 Future work

DataMole currently lacks the ability to fit pipelines on a training set and apply them to different test sets. To implement this feature, the current pipeline system could be extended with the possibility of exporting both fitted and unfitted pipelines. For interoperability with Python it may be useful to define every operation using Scikit-learn Transformer API [11], thus giving the possibility to graphically create pipelines and to later use them where needed, even in Python scripts.
Other desirable features that have not been implemented are an *undo* for transformations and the application of *custom functions* to dataframe columns. The latter can be easily implemented with a simple code editor that allows to define a Python function. More refined approaches may include the definition of a special simplified language

with support for the main constructs and operators, or alternatively support for a visual programming paradigm, like the one used in visual programming languages. Support for big data is also missing: it may be achieved by using scalable data containers, like the ones offered by packages described in §3.1.1.
Finally, DataMole multithreading support can be improved: multithreading was used to make sure that the graphical interface is not blocked while operations are being executed. The main drawback of the selected approach, described in §3.3.4.6, that makes use of `QThreadPool` and `QRunnable`, is that it does not provide the ability to stop running operations. Additionally, Python is quite limited when operating on multiple threads, since the Global Interpreter Lock (GIL) only allows for one thread to take control of the Python interpreter. Most of the operations that are done on separate threads in DataMole involve the Pandas library. Pandas does not always release the GIL, so this may be a bottleneck in some situations. Thus, multiprocessing may be a better solution to achieve real parallelism. However, as already explained, the objective of multithreading in DataMole is to avoid a frozen GUI, and multithreading was enough for this purpose.

Appendix A

Developer manual

This chapter describes how DataMole can be extended with new functionalities. It is an extension of chapter 3, that is assumed to be read before this one, since it discusses high-level architectural choices that are not repeated here.
Part of the development was dedicated to making DataMole easily extensible. Extensibility is meant with respect to the DataMole core features, which are the usage of operations and the visualisation features, like charts. Thus the following topics are discussed in this chapter:

- Definition of *new operations*, to be applied from the *Flow panel* or singularly (§A.2);
- Extension of the *View panel* with new widgets (§A.3).

Additionally the last sections describe how to use the notification system (§A.4), to visualise pop-up with custom messages, the DataMole logger (§A.5) and finally §A.6 explains how non-code files can be added to the resource system.
This chapter only describes classes, methods and everything relevant in order to extend DataMole. A complete description of the program API, with all classes and methods, will be released in the software repository along with the code.

A.1 Package organisation

DataMole is organised as a standard Python package, and its complete structure is shown in the tree below, where only the `tests/` and `docs/` folders have not been expanded.
An overview of the content of the main sub-packages, marked in green in the below tree, was given in §3.3.1.

```
dataMole/
├── main.py
├── requirements.txt
├── makefile
├── docs/ ..... Automatic documentation
│   └── ...
├── tests/
│   └── ...
└── dataMole
    ├── threads.py
    ├── status.py
    ├── resources.qrc
    ├── exceptions.py
    ├── utils.py
    ├── flow/ ..... Pipeline management
    │   ├── dag.py
    │   └── handler.py
    ├── operation/ ..... Every opeation is defined here
    │   ├── actionwrapper.py ..... Handles operation execution
    │   ├── cleaner.py
    │   ├── dateoperations.py
    │   ├── discretize.py
    │   ├── dropcols.py
    │   ├── duplicate.py
    │   ├── extractseries.py
    │   ├── fill.py
    │   ├── index.py
    │   ├── input.py
    │   ├── join.py
    │   ├── onehotencoder.py
    │   ├── output.py
    │   ├── removenan.py
    │   ├── rename.py
    │   ├── replacevalues.py
    │   ├── scaling.py
    │   ├── typeconversions.py
    │   ├── utils.py
    │   ├── readwrite/ ..... I/O operations
    │   │   ├── csv.py
    │   │   └── pickle.py
    │   ├── computations/ ..... Worker operations
    │   │   └── statistics.py
    │   └── interface/
    │       ├── graph.py
    │       └── operation.py
    ├── data/ ..... Dataframe utilities
    │   ├── Shape.py
    │   ├── Frame.py
    │   └── types.py
    └── gui/ ..... Qt GUI classes
```

```
    ├── mainmodels.py
    ├── workbench.py
    ├── window.py
    ├── utils.py
    ├── charts/
    │   ├── views.py
    │   ├── scatterplot.py
    │   ├── timeseriesplot.py
    │   ├── histogram.py
    │   └── utils.py
    ├── panels/
    │   ├── framepanel.py
    │   ├── diffpanel.py
    │   ├── viewpanel.py
    │   ├── attributepanel.py
    │   └── dataview.py
    ├── graph/
    │   ├── node.py
    │   ├── controller.py
    │   ├── constant.py
    │   ├── edge.py
    │   ├── rubberband.py
    │   ├── scene.py
    │   ├── view.py
    │   └── polygons.py
    ├── widgets/
    │   ├── waitingspinnerwidget.py
    │   ├── statusbar.py
    │   ├── operationmenu.py
    │   └── notifications.py
    └── editor/
        ├── configuration.py
        ├── interface.py
        ├── OptionsEditorFactory.py
        └── infoballoon.py
├── flogging/ ..... Logging package
    ├── loggable.py
    ├── utils.py
    └── operationlogger.py
├── resources/ ..... Static resources
    ├── style.css
    ├── descriptions.html
    └── icons/
└── config/ ..... Configuration files for extension
    ├── operations.json
    └── dataviews.json
```

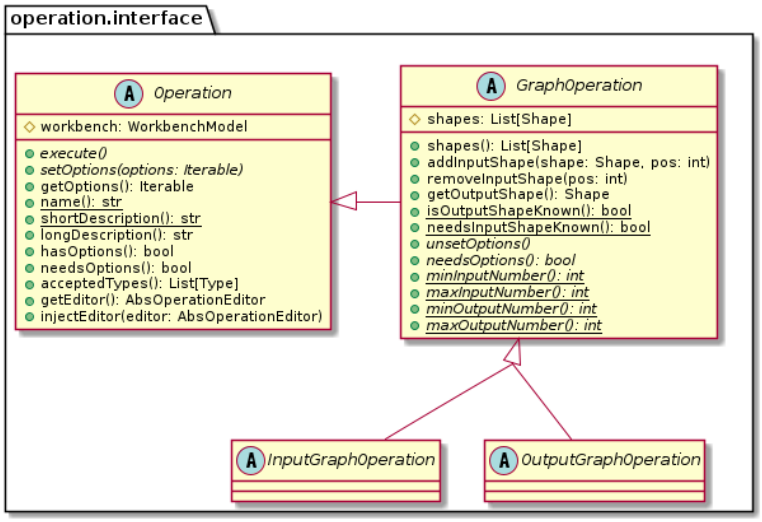


Figure A.1: Abstract classes derived from `Operation`

A.2 Definition of a new operation

Adding a new operation is simple. Three steps are required:

1. Choose the `Operation` abstract class to subclass;
2. Define the new operation inside a module;
3. Edit a configuration file to tell DataMole where to look for new operations.

When defining a new operation it is probably necessary to define an editor widget to support setting user options. This step is described later, in §A.2.4.

A.2.1 Choosing the abstract class

All data transformations in DataMole are represented with subclasses of the `Operation` base class. Package `operation.interface` contains the definitions of 4 abstract classes shown in Fig. A.1. The choice of the class to inherit depends on the operation that needs to be defined:

- `Operation`: subclass this to define an operation as a generic task. The only abstract methods are `execute` and `setOptions`. Such an operation can not be used with the pipeline in the *Flow panel*, but can only be applied from the *Attribute panel*. Even background workers can implement this interface to take advantage of the multithreading module, as is later described in §A.2.5;
- `GraphOperation`: its subclasses can be used in the *Flow panel* as well as in the *Attribute panel*, provided it has only one input. This constraint is set to reduce complexity, since graph operations must be adapted to be used from the *Attribute*

panel and having to deal with multiple inputs was not easy.
Graph operations must not do side effects on their inputs (or on the workbench);

- `InputGraphOperation`: this is defined for convenience. It gives a default implementation to some methods that are not relevant when defining an input operation for the pipeline graph. These operations has no input nodes;
- `OutputGraphOperation`: again this is subclassed only once by the graph operation `ToVariable` which has no output and exactly one input. This interface may be used to define an operation with no outgoing edges.

Every operation can hold a reference to the workbench object, but this is optional. In general `GraphOperation` subclasses should avoid any side effect: the pipeline consists of a chain of *functional* transformations where the output of the parent operation is set as the input of the child node, and every operation must not change its input while producing its output. `InputGraphOperation` and `OutputGraphOperation` can instead do side-effect on the workbench: in fact they are mainly used to retrieve the input dataframe from the workbench when the pipeline starts and to save the pipeline output when it completes.

A.2.1.1 A comment about subclassing in Python

Interfaces and abstract classes are used in DataMole to enforce the presence of specific methods in subclasses, that are required for them to correctly work. But the type information itself (e.g. the mere fact that an operation is a subclass of `Operation`) is not actually important, because Python is dynamically-typed and uses the *duck typing* approach, thus subclasses type is not checked. As a consequence, interface realisation is never strictly required, provided that all the relevant methods are defined. For instance an operation can be any object that implements all the methods described in §A.2.2, even if it is not a subtype of `Operation`.

A.2.2 Implementing the operation

New operations can be defined in new modules inside the `operation` package, where all DataMole operations are defined. Additionally related transformations are usually grouped in the same file. For example the module `type.py` contains the definition of all the operations used for type conversion. However it is not important where the new operations are defined, so they may be stored in different packages if desired.
The `operation` package also contains an `utils` module with many helper functions for parsing editor options and some validators used by the editor widgets to validate inputs.

A.2.2.1 *Operation* methods

This section describes the methods that every `Operation` subclass inherits and override when required.

- `execute(*args, **kwargs)`: this method is run to execute the operation. It takes any number of arguments (0 as well) and returns whatever the operation produces. It may also return nothing, if the operation does side effects;
- `setOptions(*args, **kwargs)`: the method used to configure an operation with its options. This method is called with the arguments provided by the `getOptions` method of the editor widget, so its arguments depends on the way the widget provides options. For example if the editor widget returns a tuple of 3 integers, this method should expect to receive 3 integer arguments. If the operation does not require options this method can be set to a no-op. This method can also perform fields validation, see §A.2.2.2;
- `getOptions()`: this method returns the options currently set in the operation, in the same format required by `setOptions`. If you are defining an operation to use from the *Attribute* panel this method is probably never used and its reimplementation can be skipped. The default implementation returns an empty dictionary;
- `name()`: this static method returns a string with the name of the operation to be shown to the user;
- `shortDescription()`: a static method with a reasonably short description to be visualised inside the header of the editor widget;
- `longDescription()`: returns the text to be shown when the user clicks the "More" button in the editor widget. The description can be long and can include HTML formatted text. To avoid cluttering code with long formatted strings, they can be placed in the `resources/descriptions.html` file, under a `section` tag with the operation class name. By default this method searches the description in that file. More details are in §A.6;
- `hasOptions()`: returns a boolean value saying whether all the required options are set. This is required because every operation can have its own option fields. Typically it returns *True* if all option fields are not set to *None*;
- `needsOptions()`: returns a boolean value that tells whether the operation needs options, and consequently an editor widget. If this method returns *False* the `getEditor` method can be no-op;
- `acceptedTypes()`: returns the list of types that the operation supports, to choose between Numeric, Nominal, Ordinal, String and Datetime. These types are defined in the `data.types` module. By default it supports all types;

- `getEditor()`: builds the editor widget of type `AbsOperationEditor` which should be used to configure the operation. This is described in §A.2.4;
- `injectEditor(AbsOperationEditor)`: if some editor components require additional configuration that cannot be provided during the editor creation, this method can be reimplemented. Typically it is used to fix column size on tables created with the editor factory.

A.2.2.2 Options validation

Operations can refuse to accept parameters if they are not set correctly: options validation may be performed inside the `setOptions` method of every operation. This method should first check if the provided options are acceptable and save them only if they are. Otherwise it should raise an exception of type `OptionValidationError`. This exception can be parametrised with a list of errors that occurred while validating the options, thus it allows to notify more than one error at once. Every item of this list is a pair made up of a string error code and an error message. The error code allows to customise error handling and will be discussed in §A.2.4.1.
To give a practical example, part of the definition of the `BinsDiscretizer` class is reported here and commented:

```
1  class BinsDiscretizer(GraphOperation, Loggable):
2    def __init__(self, *args, **kwargs):
3      super().__init__(*args, **kwargs)
4      # Initialisation of the options fields
5      # Strategy to use for discretizing
6      self.__strategy: BinStrategy = BinStrategy.Uniform
7      # The column indices to be discretized
8      self.__attributes: Dict[int, int] = dict()
9      # The suffix for the column to create (if the
           transformation is not done in-place)
10     self.__attributeSuffix: Optional[str] = '_discretized'
11
12   def setOptions(self,
13     attributes: Dict[int, Dict[str, str]],
14     strategy: BinStrategy,
15     suffix: Tuple[bool, Optional[str]]) -> None:
16     # 'attributes' is a dictionary like {row: {column\_key:
           value} }
17     # Thus it maps every row to the values of any column in
           the table
18
19     # Validate options
20     errors = list()
21     if not attributes:
```

```
22        # Error: the user did not select any attribute
23        errors.append(('e1', 'Error: At least one attribute
              should be selected'))
24      for r, options in attributes.items():
25        bins = options.get('bins', None)
26        if not bins:
27          # Error: column 'bins' is not set for this row
28          errors.append(
29            ('e2', 'Error: Number of bins must be set at row {:
                  d}'.format(r))
30            )
31        elif not isPositiveInteger(bins):
32          # Error: column 'bins' is not a valid number
33          errors.append(('e3', 'Error: Number of bins must be >
                1 at row {:d}'.format(r)))
34      if strategy is None:
35        # Error: no strategy is selected from the radio buttons
36        errors.append(('e4', 'Error: Strategy must be set'))
37      if suffix[0] and not suffix[1]:
38        # Error: suffix is not set
39        errors.append(('e5', 'Error: suffix for new attribute
              must be specified'))
40      if errors:
41        # If any validation error occurred stop
42        raise OptionValidationError(errors)
43
44      # No error occurred, then set options
45      # Clear previously set attributes
46      self.__attributes = dict()
47      # Set options
48      for r, options in attributes.items():
49        k = int(options['bins'])
50        self.__attributes[r] = k
51      self.__strategy = strategy
52      self.__attributeSuffix = suffix[1] if suffix[0] else None
```

In the above example the `setOptions` method checks for options correctness in lines 20-39. Then if one or more validation errors were detected it raises an exception (line 42), otherwise it sets the new options (lines 44-52). The error message will be shown to the user as in Fig. A.2.

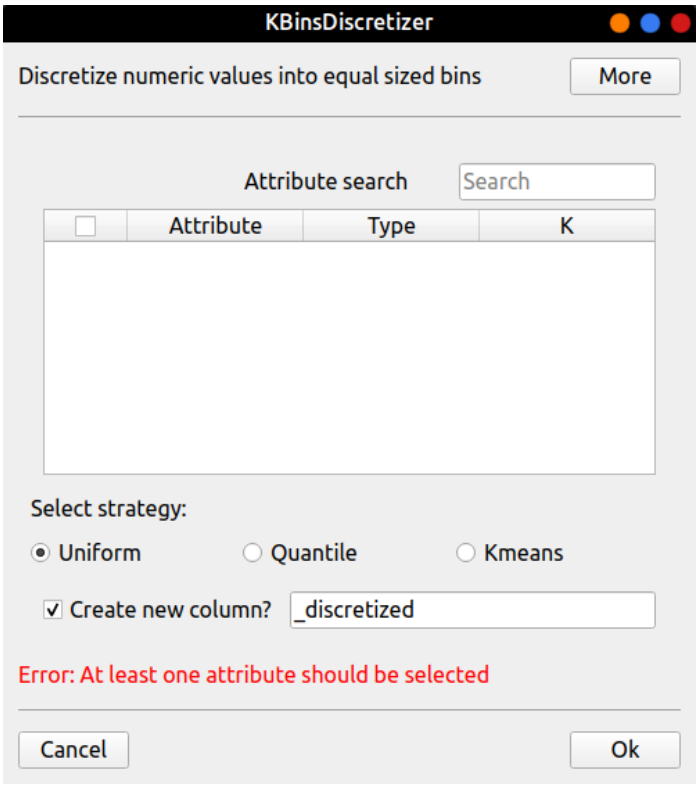


Figure A.2: Error message in the *BinsDiscretizer* operation

A.2.2.3 *GraphOperation* methods

The following section describes methods defined in the `GraphOperation` class. All methods inherited from `Operation` are not discussed here, unless they require further explanation.

A.2.2.3.1 Utility methods

The following three methods provide functionalities required to manage every operation inside the pipeline. They are already implemented and should not be redefined in most cases.

- `shapes()`: getter method for the protected `shapes` field, which contains the shapes of the inputs of the operation;
- `addInputShape(Shape, int)`: this method is used to set the shape at the specified position in the `shapes` list;
- `removeInputShape(int)`: remove the `Shape` object at the specified position in the `shapes` list and sets it to `None`.

A.2.2.3.2 Methods to be reimplemented

The following methods should be overridden in every subclass to customise the operation behaviour.

- `execute(*Frame)`: the execute method has a different signature from the one defined in the `Operation` class. As already explained graph operations should be defined in a functional way, with no side effects. Hence this method is passed the input dataframes and must apply the transformation and return the output dataframe (of type `Frame`) without affecting its input;

- `getOutputShape()`: this method should be overridden to return a `Shape` object with the column names and types of the output of this operation. If the input shapes are not set (for example because the node is not yet connected to a predecessor) or any relevant option is not configured it must return `None`;

- `isOutputShapeKnown()`: this static method must return `True` if `getOutputShape` is able to infer the output shape, given the operation options and the shapes of its input. Otherwise it must return `False`. There are situations in which this method could *always* return `False`. For instance, this is the case with the operation to remove all columns with more than a threshold of NaN values: there is no way of knowing in advance which columns will satisfy this condition, hence the operation does not know its output shape;

- `needsInputShapeKnown()`: another static method that returns `True` for operations that require their input shapes to be used. Operations that do not need the input shape are the only operations that can be placed after operations that do not know their output shape (i.e. their `isOutputShapeKnown` method always returns `False`);

- `unsetOptions()`: this method resets every option that depends on the input shapes of the operation. It is called by the `DagHandler` whenever new input shapes are propagated through the pipeline;

- `minInputNumber()`: a static method which returns the minimum number of input connections that the operation supports;

- `maxInputNumber()`: a static method returning the maximum number of input connections that the operation supports or -1 if there is no maximum;

- `minOutputNumber()`: a static method returning the minimum number of output connections that the operation supports;

- `maxOutputNumber()`: a static method returning the maximum number of output connections that the operation supports or -1 if there is no maximum;

A.2.3 Export the operation

Once operations have been defined it is necessary to make them visible to DataMole. Every module must define a global variable `export` pointing to the new operation class or, if more than one operations are defined in the same module, to a list (or tuple) of classes. For instance if two new operations were defined in the same module with classes `TransformData1` and `TransformData2`, the export variable should be set as in this snippet:

```
class TransformData1(Operation, Loggable):
  def execute(self, *args, **kwargs):
    pass
  ...

class TransformData2(Operation, Loggable):
  def execute(self, *args, **kwargs):
    pass
  ...

class OtherStuff:
  # Class that is not an operation
  ...

export = TransformData1, TransformData2
# or export = [TransformData1, TransformData2]
```

Additionally the name of every module to be searched for operations must be appended to the list defined in file `config/operations.json`. The fully qualified name of the module should be used, like in the following example:

```
{
  "modules": [
    "dataMole.operation.fill",
    "dataMole.operation.discretize",
    ...
    "dataMole.operation.myNewModule"
  ]
}
```

A.2.4 Definition of editor widgets

Every operation requiring user options is responsible for defining its specialised *editor widget* to support options configuration. An example is in Fig. A.3.
Every editor widget must subclass the `AbsOperationEditor` abstract class, defined in the `gui.editor` package. Package structure is shown in Fig. A.4.
Custom editors are usually defined in the file containing the operation definition, since

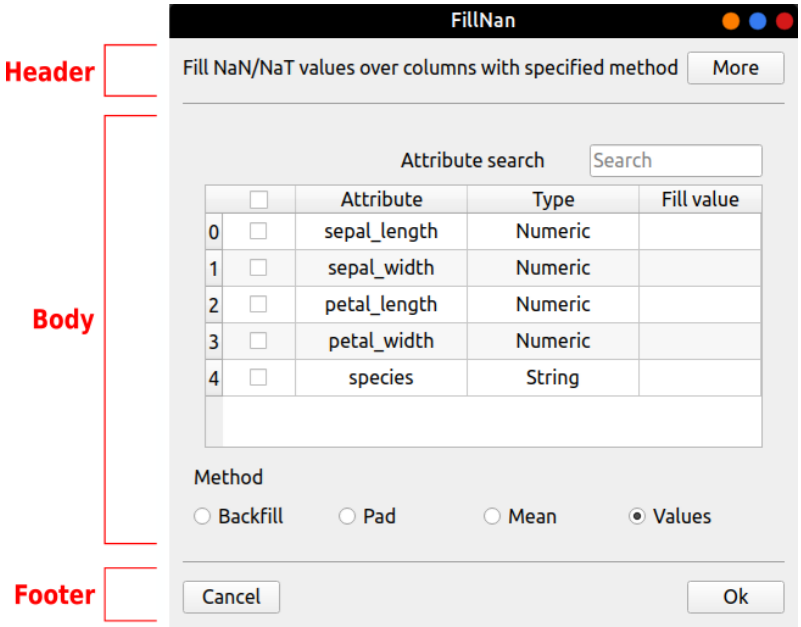


Figure A.3: The three parts that compose every *editor widget*. The *header* shows the short description and button to access the long description of the operation. The *footer* has a button to quit the editor and one to confirm the options set.

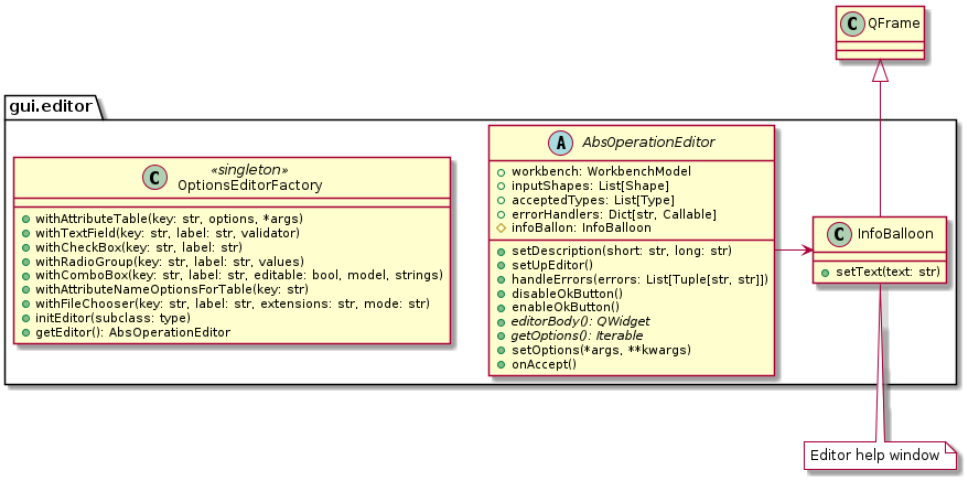


Figure A.4: Classes defined in the `gui.editor` package

they are used only by a single operation. However, they can be placed on a different file if they are reused many times.
An *editor widget* should reimplement the following methods:

- `editorBody()`: this abstract method should return the `QWidget` to place in the editor, between the header and the footer;
- `getOptions()`: this abstract method must return the options set in the editor. It should return them inside an iterable sequence that can be unpacked when passed to the `setOptions` method of the operation. Its return type should therefore be compatible with the argument type of the operation `setOptions` method;
- `setOptions(*args, **kwargs)`: sets the options inside the editor. The default implementation does nothing, which is ok in case the operation does not require a configurable editor;
- `onAccept()`: this hook method is called after the options are confirmed (i.e. the *"Ok"* button is clicked) and before setting them in the operation. It can be reimplemented to perform additional actions. Does nothing by default.

A.2.4.1 Customised validation error handling

As previously described, when the user configures an editor widget, the operation can raise an error with type `OptionValidationError` if supplied options are not correct. In this situation, every editor widget shows the error messages in red font immediately above the footer, like in Fig. A.2. This is generally good enough, but complex editors may behave differently, for example by applying a red border around a `QLineEdit` or by showing pop-up immediately above the wrong fields. In fact, editor widgets derived from `AbsOperationEditor` support custom error handling behaviours. This abstract class defines a `errorHandlers` public field, of type `Dict[str, Callable]`, that contains arbitrary functions (`Callables`) used to handle specific validation errors. Recall from §A.2.2.2 that every pair passed to the `OptionValidationError` constructor contains an *error code*, as well as an error message. When handling validation error, the editor first checks if a custom error handler is provided: to do this it searches the `errorHandlers` dictionary for a pair with the specified error code as key. If a matching key is found, the corresponding method is invoked, otherwise the default error handling strategy is used.
Hence, once a customised method to handle errors has been defined, it may be added to this dictionary with an error code that matches the one of the error it should handle.

A.2.4.2 The editor factory

Since many operation editors required very similar components a *factory class* has been defined to quickly build standard editors with a variety of fields. To do this, the factory class `OperationEditorFactory` can be instantiated inside method

`Operation.getEditor` and used to configure the editor. Since the factory is a *singleton*, method `initEditor` must be always called to initialise a new editor.
When the factory is used, widgets options are passed around between the editor and the operations as Python dictionaries. This is the reason why methods `getOptions` and `setOptions` of classes `AbsOperationEditor` and `Operation` also accept key-value pairs with the `**kwargs` argument. The key used for every option can be specified with the `key` argument when using the factory methods described below.

- `withAttributeTable(key, options, checkbox, nameEditable, showTypes, types, *args)`: adds a table to show dataframe columns, with optional column of checkboxes for selection and any number of additional columns. For example the widget in Fig. A.3 had a table with 1 additional column, named *Fill value*, to specify the value that should be used to substitute NaNs. The `options` argument is a dictionary to specify the additional columns to show in the table: every entry consists of a column identifier and a tuple with the column name, a delegate for that column and a default value to show when nothing is set. `checkbox`, `nameEditable`, `showTypes` are boolean parameters to control whether to show a checkbox column, whether the name column should be editable and the type column should be showed. Argument `types` allows to filter only certain types, in case the operation does not support every type. For example the *Fill NaN* editor table is configured with this method call:

```
1   factory.withAttributeTable(
2             key='selected',
3             checkbox=True,
4             nameEditable=False,
5             showTypes=True,
6             types=self.acceptedTypes(),
7             options={
8               'fill': (
9                     'Fill value',
10                    OptionValidatorDelegate(
                          SingleStringValidator()),
11                    None
12                  )
13                })
```

  The *delegate* is Qt component that controls how column items are rendered inside the view. By defining a custom delegate, it is possible to change items appearance in any way: the checkbox in the first column of Fig. A.3, for instance, is created by defining a custom delegate for boolean values. The `OptionValidatorDelegate` was defined for convenience and only provides a customised validation the input, through the `QValidator` passed as its argument. If no delegate and no default value is needed, `None` can be used in their place. Definition of custom delegates will not be discussed here, since it is part of the Qt Framework and is explained

in its official documentation;

- `withTextField(key, label, validator)`: adds a `QLineEdit` setting a label above it and with an optional `QValidator` for its input. Some validators commonly used are defined in the `operation.utils` module. Widget marked with *(T)* in Fig. A.5 was created using this method;

- `withCheckBox(key, label)`: adds a `QCheckBox` with a label;

- `withRadioGroup(key, label, values)` inserts a group of `QRadioButtons` with the specified label above it. the `values` argument is a list of pairs, that maps the label to show (as a string) with the combo box value (of any type). The radio buttons in Fig. A.3 is created with the following options:

```
1   factory.withRadioGroup(
2           key='fillMode',
3           label='Method',
4           values=[
5                ('Backfill', 'bfill'),
6                ('Pad', 'ffill'),
7                ('Mean', 'mean'),
8                ('Values', 'value')
9           ])
```

- `withComboBox(key, label, editable, model, strings)`: adds a `QComboBox` with a label. If `editable` is *True* the combo box allows to enter arbitrary values, otherwise it only allows to choose between one of the predefined values. Arguments `model` and `strings` allow to set the values to show when the combo box is used. A Qt model class can be used or alternatively a list of strings can be provided;

- `withAttributeNameOptionsForTable(key)`: using this method allows to add a widget like Fig. A.6. It adds an option to avoid overwriting attributes when transformations are applied by defining new ones with the specified suffix;

- `withFileChooser(key, label, extensions, mode, **kwargs)`: used to create a window that allow to choose an existing file using a `QFileDialog`. Showed files can be filtered by their extension using the `extensions` argument. The `mode` string argument must be set to `"save"` or `"load"`, depending on whether the file dialog should allow to select non existing files (*save* mode) or not (*load* mode). An example of such widget is shown in Fig. A.5, marked with *(F)*. Additional arguments can be passed to the `QFileDialog` by setting them as `**kwargs`;

- `initEditor(subclass)`: this method must be called before any factory method, and is used to initialise a new editor widget. The factory is a singleton, thus a call to this method resets its internal state and clean the parameters of widgets previously created. The `subclass` parameter accepts a `type` of a class that should

be used as base class for the new widget. Of course it must be a subclass of `AbsOperationEditor`;

- `getEditor()`: assembles the new editor widget and returns it.

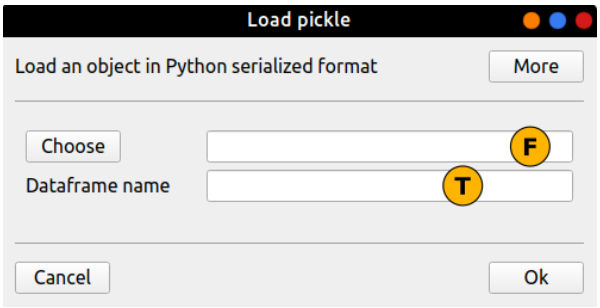


Figure A.5: Widget to import *pickle* dataframes created with factory methods

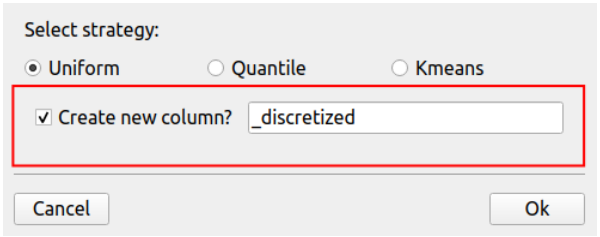


Figure A.6: Widget created with factory method `withAttributeNameOptionsForTable`

Using the factory, a typical `getEditor` implementation for an operation may look like this:

```
1 def getEditor(self):
2   factory = OptionsEditorFactory()
3   factory.initEditor()
4   ...
5   # Call factory methods
6   ...
7   return factory.getEditor()
```

Finally it must be said that editor widgets defined using the factory do not support the customised error handling mechanism described in §A.2.4.1.

A.2.5 Creating worker operations

Sometimes there is the necessity to do some background computation without freezing the user interface. Operations can also be defined for completing tasks that can be computed in background. For example the `operation.computations.statistics` module contains two operations used respectively to compute data for the *statistics panel* and the *histogram* when an attribute is clicked in the *Attribute* tab.
The `threads` module can then be exploited to run the operation in another thread. This module defines a `Worker` class (deriving from `QRunnable`) that can be scheduled for execution in a `QThreadPool`.
After defining a worker operation by implementing the `Operation` interface, a background computation can be set up using this general pattern:

```
1  # Define a worker operation
2  class BackgroundComputation(Operation):
3    def __init__(*args):
4      # Initialise
5      ...
6
7    def execute(arg1, arg2, *args):
8      ...
9      # Computation
10     ...
11     return result
12
13 # Initialise a new operation
14 comp = BackgroundComputation(*my_args)
15 # Set eventual options and define arguments for the execute()
       method
16 myArg1 = ...
17 myArg2 = ...
18 # Create a worker for the operation and set the execute()
     args
19 worker = Worker(comp, args=(myArg1, myArg2), identifier='
     comp1')
20 # Connect worker signals to appropriate handler slots
21 worker.signals.result.connect(self.onResult)
22 worker.signals.error.connect(self.onError)
23 worker.signals.finished.connect(self.onFinish)
24 # Start computation on the thread pool
25 QThreadPool.globalInstance().start(statWorker)
```

The `worker` needs an identifier that will be passed back as the first parameter of the emitted signals. This identifier can be of any type, and should be used to recognise which worker emitted a particular signal, but can be omitted if it is not relevant.

The `args` parameter can be omitted as well if the operation `execute` method does not require arguments. In the above example `onResult`, `onError` and `onFinish` are methods marked as Qt *slots* (which allow to connect them to signals) and are invoked when the worker status changes, through the following signals:

- `result(id: object, result: object)`: this signal is emitted when the worker completes successfully (i.e. without runtime errors) and carries two arguments, the identifier passed to the worker constructor and the value returned by the `execute` method, which can be `None` if the operation does side effects with the result;
- `error(id: object, err: tuple)`: it is emitted when the `execute` method fails with a runtime error. The second parameter is a tuple with the type of the exception, the exception object itself and the stack-trace as a string. This data can be used to create a log entry and to notify the user;
- `finished(id: object)`: signal emitted after the worker stops executing, either because it failed (and the `error` signal was emitted) or because it completed successfully (and the `result` signal was fired).

A.3 Extension of the *View panel*

Currently the *View panel* supports the creation of two type of charts, the line chart for time series and the scatterplot matrix, but it is possible to add customised visualisation features to this panel. The active widget can be switched by using the combo box shown in Fig. A.7. These widgets are dynamically discovered and loaded every time DataMole is started by looking at the configuration in file `config/dataviews.json`. This operation is done in the `__init__` file of the `gui.panels` package. The *json* file contains the following lines:

```
{
  "config": {
    "default": "Scatterplot",
    "description": "Select a data view:"
  },
  "classes": {
    "Scatterplot": "dataMole.gui.charts.scatterplot.
       ScatterPlotMatrix",
    "Time series": "dataMole.gui.charts.timeseriesplot.
       TimeSeriesPlot"
  }
}
```

The `classes` dictionary contains the fully qualified name of the widget class to show in the panel, with the label to show in the combo box as keys. The `config` dictionary contains the label to set as default one in the combo box, and the label to place before

the combo box.
Hence, in order to add widgets to this panel, one should:

1. Define the new widget implementing the `DataView` interface; it requires the definition of a single *slot*, namely `onFrameSelectionChanged`,in order to react properly when the user changes the active dataframe;

2. Add the new class name to the `classes` dictionary in the `config/dataviews.json` file.

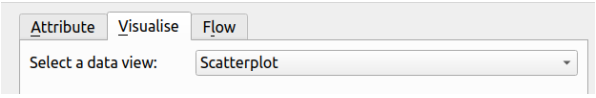


Figure A.7: Combo box used to switch active widget in the *View panel*

A.4 Using the notification system

Small pop-ups are used to notify the user whenever an error occurs or some operation is completed. An example is shown in Fig. A.9. Notification messages are stacked vertically and can be closed by clicking on the small right button.
These pop-ups are defined in the `gui.widgets.notifications` module, which contains 3 classes, outlined in diagram Fig. A.8. An instance of the `Notifier` class is always available in the *gui.notifier* global variable. Thus in order to add notifications, this global variable should be imported from the `gui` package. The `addMessage` method can be used to add messages, or they can be cleared invoking `clearMessages`.
The `gui` package also exposes the `gui.statusBar` variable, which is the global access point to the DataMole status bar, placed at the bottom of the main window, shown in Fig. A.9. It inherits `QStatusBar`, so its methods can be used to show messages.

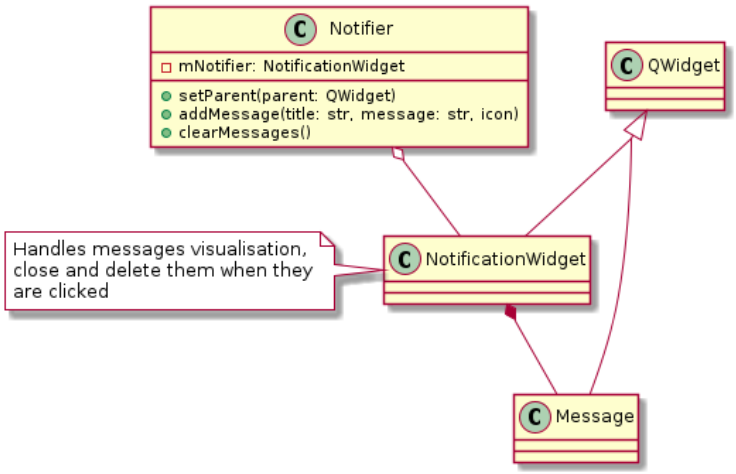


Figure A.8: Class diagram of the `notifications` module

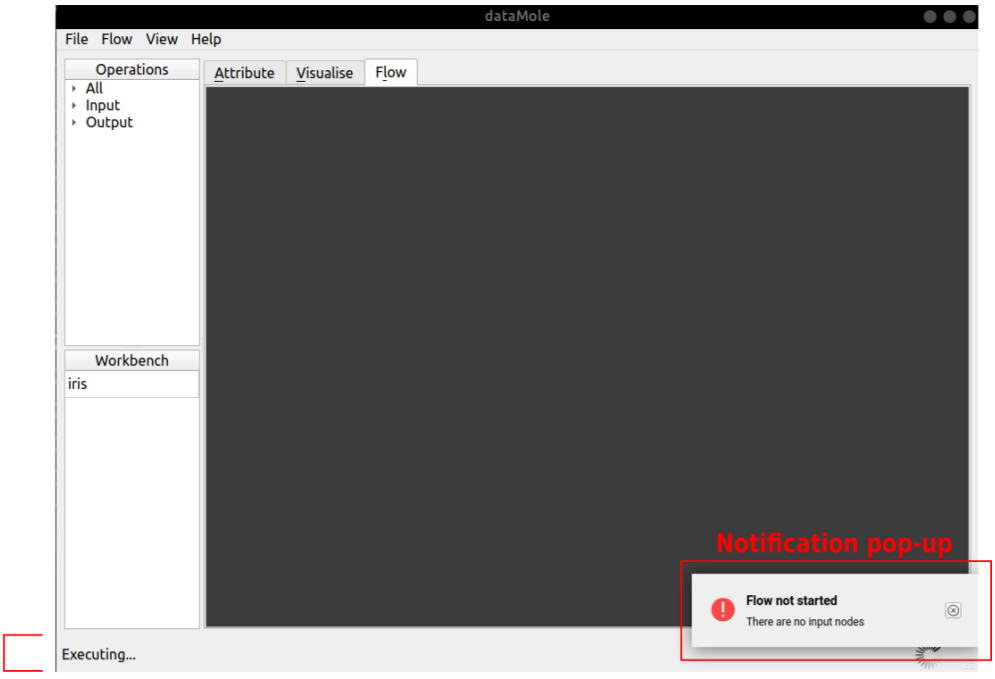


Figure A.9: The main window, with a message on the status bar and a pop-up used for notifications. In this example the user tried to execute a pipeline with no input nodes.

A.5 The logging package

DataMole defines a collections of logging utilities in the `flogging` package, which makes used of the Python `logging` module. Three loggers are defined:

- `appLogger`: the *application logger*, where debug messages are printed, and the standard error stream (`stderr`) is redirected. These log files are created in the `logs/app` folder;
- `graphLogger`: when a pipeline is executed the *DagHandler* creates a new log file in the `logs/graph` folder and uses this logger to log every operation. In order to be logged operations need to have implemented the `Loggable` interface, defined in the `flogging.loggable` module;
- `opsLogger`: every operation applied singularly (i.e. outside of the pipeline) is logged through this object inside the `logs/operations` folder. Also in this case the operations need to implement the two methods of the `Loggable` interface.

Loggers are objects of type `logging.Logger` and can be imported directly from the `flogging` package. A simple example is the following:

```
1 from dataMole import flogging
2
3 # Log to the application logger
4 flogging.appLogger.warning('A warning')
5
6 # Log to the operation logger
7 flogging.opsLogger.error('An operation failed')
```

A.5.1 Implementing the *Loggable* interface

The two methods defined in `Loggable` interface, that must be implemented in order to log an operation, are the following:

- `logOptions`: must return a formatted string with the configuration of the operation, like its options. This method is invoked before the operation is executed;
- `logMessage`: returns a formatted string with additional info. Differently from the previous method, this one is invoked after the operation completes, but it is not called if the operation fails with an error. It may include details on what happened during execution. For instance, in the `BinsDiscretizer` class it is used to inform the user of which intervals were used for discretization.

A.6 The resource system

The Qt resource system is a platform-independent mechanism for storing binary files in the application's executable. This is useful for applications that need to access a certain set of files like icons, translation files, etc. [18]. The `resource` directory groups every such file used in DataMole. It has the following structure:

```
dataMole/
├── resources.qrc ..... Qt resource file
└── resources/
    ├── icons/ ..... PNG icons
    ├── descriptions.html ..... Operations descriptions
    └── style.css ..... Stylesheet
```

A.6.1 The operation description file

File `descriptions.html` contains the formatted text that is returned by method `longDescription` of an operation. Since these descriptions can be quite long, and need to be formatted properly using HTML syntax, I decided to put them all in single file, instead of having them scattered inside the operation classes. It is possible to add new descriptions or edit the existing ones directly in this file. Every new description must be placed in a new section, with a `name` attribute set to the class name of the related operation, like in this example:

```
<section name="MyOperationClass">
  <h2>Operation name</h2>
  Very short description <br/>
  <h3>Options description</h3>
  ...
  Explain how the operation can be configured
  ...
</section>
```

There are no rules on how to write the long description, but it should include everything that is needed to understand the purpose of the operation and how it should be configured, if some options are required.
The description file is read during initialisation of the `operation` package, inside the `__init__` file.

A.6.2 Adding new resources

Adding a new resource can be done by adding its path to the `resources.qrc` file, as explained in the official Qt documentation. After that, the resources must be converted to bytecode by using the Qt Resource Compiler, which, in Qt for Python, can be invoked with the `pyside2-rcc` command.
For convenience DataMole comes with a makefile that include the command needed to do this: it is sufficient to run "`make resources`" from the main folder. This command will generate a file named `qt_resources.py` with the bytecode for every resource.

Appendix B

DataMole user manual

This guide describes how to use DataMole for data analysis. The tool allows to load tabular datasets from disk and apply transformations to their columns. This manual does not include a list of available transformation, nor does it explain how to use them, since this information can be accessed directly while using DataMole through a specific help widget.
A section describing how to install and launch DataMole was omitted, since this information is kept updated in the `readme` file of the GitHub repository.

B.1 Using DataMole

B.1.1 Importing a dataset

DataMole can import tabular datasets from `CSV` and `pickle` files that contain serialised Pandas dataframes. Notice that `pickle` files may contain any Python object, but only Pandas dataframes can be loaded in DataMole.
By clicking `File > Import > "From csv"`, the editor shown in Fig. B.1 appears. It allows to choose the file separator and to select which columns to load. Columns can be selected in the provided table, that, depending on the size of the dataset, may require some time to be shown. Big datasets can be loaded in multiple dataframes, by selecting *"Split file by rows"* and specifying the maximum number of rows per dataframe. Clicking the *"More"* button opens a side panel with additional information on the operation.
Similarly, the widget to load a `pickle` dataframe can be opened clicking `File > Import > "From pickle"`.
Every imported dataset will be visible in the *workbench*, the widget that lists all loaded dataframes, visible in Fig. B.3.

B.1.2 Exporting a dataset

Loaded dataframes can be exported in `CSV` or `pickle` files. The latter option is useful to continue working on the dataset outside of DataMole, because every Python script

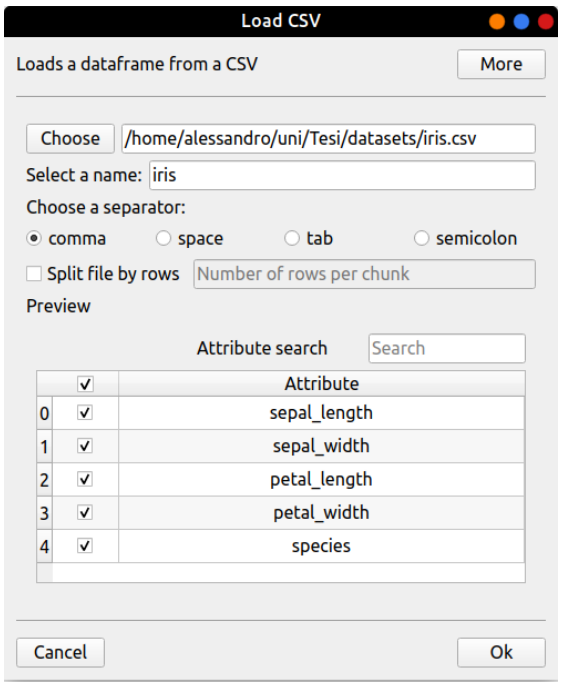


Figure B.1: The widget used to load a CSV file

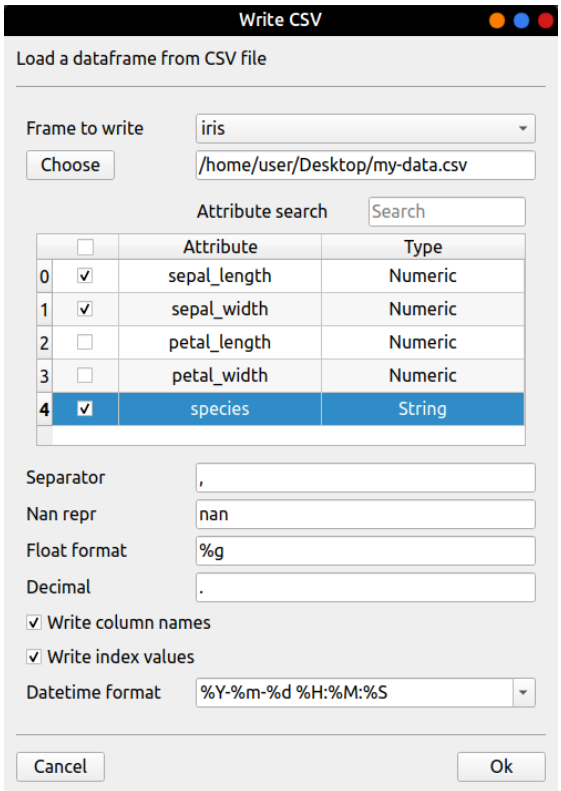


Figure B.2: The widget used to export a dataframe in CSV file

Figure B.3: The DataMole main window, with an empty *Attribute panel* on the right

can be used to load these files.
Fig. B.2 shows the widget for `CSV` export. DataMole can load multiple dataframes, thus it is required to select which one to export using a combo box. Clicking on *"Choose"* opens a dialog window for selecting the save path location. The attributes to be included can be selected in the table, and many common options can be set.

B.1.3 The main window

Fig. B.3 shows the DataMole main window, set on the *Attribute panel*, which is described later. On the right side of the window three tabs (P) can be selected to change the active panel.
On the left side, the *workbench* (W) displays the list of loaded datasets. They can be renamed by double clicking a row and typing the new name, which must be different from the name of other dataframes. Right-clicking a row opens a small context menu that allows to export and delete a dataframe. Selected dataframes can also be removed by pressing the `Canc` key while they are focused.
Above the workbench, a widget displays information on the active dataset (I) and allows to apply transformations using the menu in (A). This process is also described in the next subsection.
When an operation is executed the *status bar* (S) is used to report its status.
Three widgets can be contained in the right side of the window (C):

- The *Attribute panel*;
- The *View* panel (in the *Visualise* tab);
- The *Flow* panel.

Each one is described respectively in §B.1.4, §B.1.5 and §B.1.6.

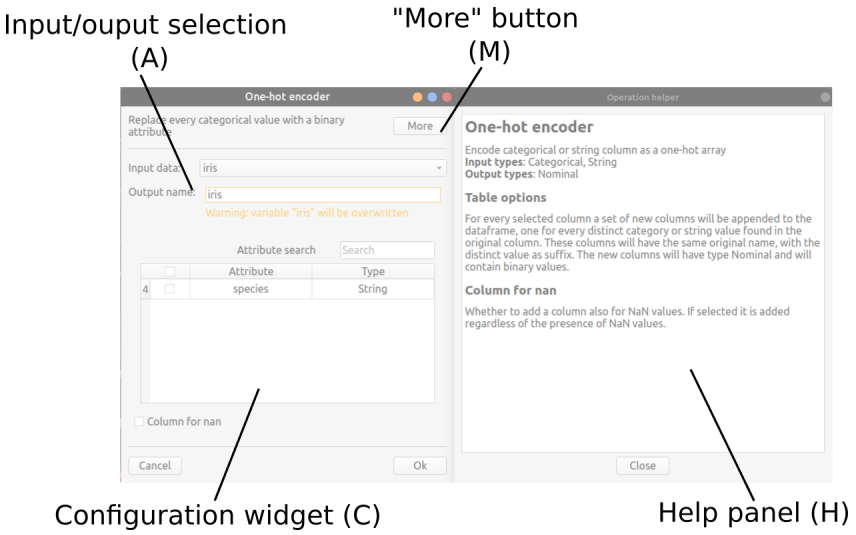


Figure B.4: The *editor widget* used to configure the one-hot encoder and its help window

B.1.3.1 Applying operations

When an operation is selected in (A) and the *"Apply"* button is pressed, the operation *editor widget* is opened, like the one shown in Fig. B.4, used to one-hot encode columns. Every editor has an header with a brief description of the operation and a *"More"* button on the right (M). By clicking on it, an help window with information on the operation and on the required parameters will be opened (H).
Below the header, a combo box allows to select the dataset to transform and a text box can be used to insert the name of the output dataset (A). After the options are confirmed with the *"Ok"* button the operation will be applied and its result will be written in a *workbench* variable with the specified output name. The output name defaults to the input name, thus the transformed dataset will replace the original one if the name is not edited.
Depending on the operation many options may need to be configured in (C). In the example the operation only requires to select the columns to encode and whether or not NaN values should be considered.

B.1.4 The Attribute panel

This panel is displayed on the left side of Fig. B.5, positioned inside the tabbed widget. It provides some basic features to get an overview of the content of a dataset.
The *attribute table* (T) displays the column names and their types: every row represents a column (i.e. an attribute) of the dataset. Column names can be changed by double clicking the cell and typing in the new name. All column names should be unique, so duplicated names will be rejected.
Above the table a search bar allows to search through attributes. Search by regular

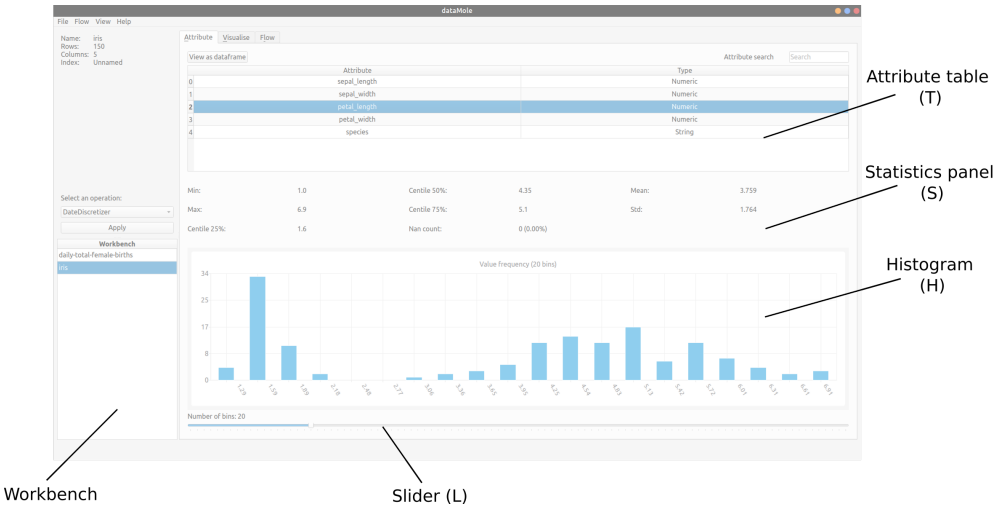


Figure B.5: The main window set on the *Attribute panel*

expression is also supported using Perl-style syntax. The search is case-insensitive.
The *statistics panel* (S) and the *histogram* (H) show information about the currently selected column, that can be changed by clicking on a row of table (T). The histogram shows the number of occurrences of every distinct values for *string* and *categorical* attributes, while *numeric* or *datetime* are first discretized in a predefined number of equal-sized intervals. This number can be changed by moving the slider below the chart (L).

B.1.5 The View panel

This tab groups DataMole visualisation features. Currently it supports the creation of a scatterplot matrix, to inspect feature correlation, and a line chart to plot time series.

B.1.5.1 Scatterplot matrix

A scatterplot matrix with 3 attributes is shown in Fig. B.6. A dataset must be selected in the *workbench* and the attributes to plot can be chosen in widget (B). Scatterplot dots are usually coloured differently depending on the values of a target attribute, that can be chosen using the combo box below the table (C).
Double clicking a scatterplot opens it in a new window, like in Fig. B.7. Here dots can be hovered to inspect their values and the chart can be zoomed and stretched as required. The content of the window can additionally be saved as an image in different formats (`PNG`, `JPEG`, `BMP`, `XMP`).
The combo box in (A) allows switching between the two chart types.

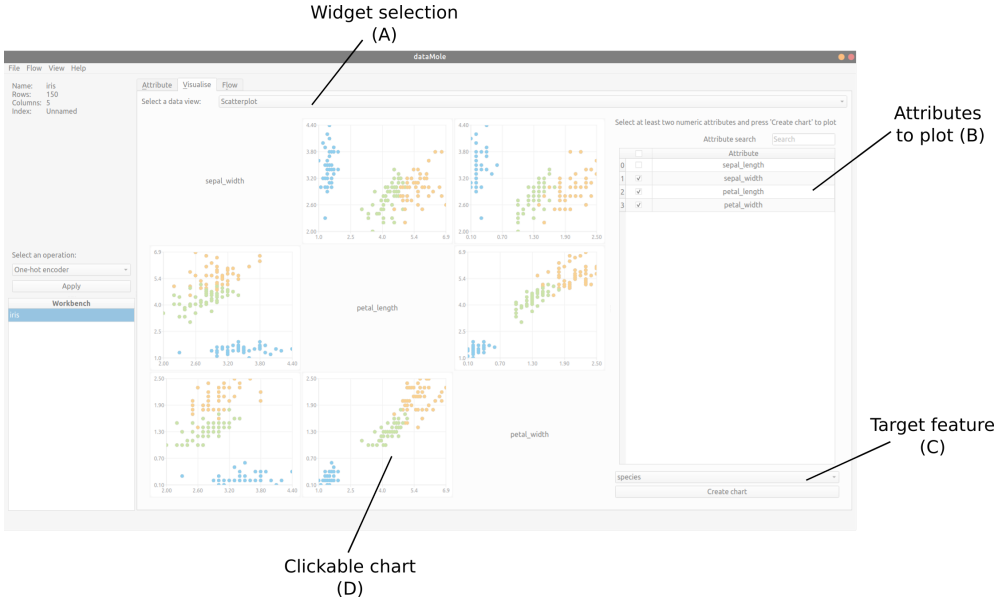


Figure B.6: A scatterplot matrix with 3 attributes

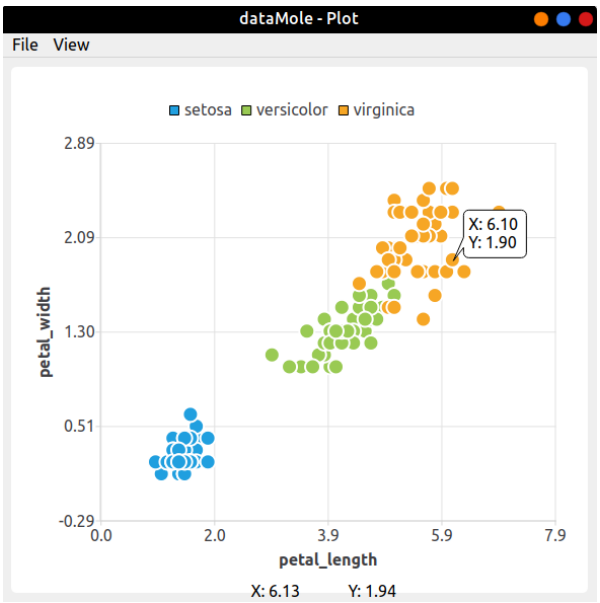


Figure B.7: A scatterplot displayed in a new window

Figure B.8: A time series displayed with a line chart

B.1.5.2 Time series plot

Time series can be represented in a line chart like in Fig. B.8 using this widget. First, the dataset must be selected in the *workbench*. Then the attribute that contain the time axis labels must be selected in (A): it must be either an attribute with type *datetime* or with *ordinal* type, since a order must be defined between its values. If a *datetime* attribute is selected the label format for visualisation in the horizontal axis can be changed in (F).
The time dependent attributes can then be selected in table (V). The chart area (P) is interactive and can be zoomed and moved around. The initial state of the chart can be restored by using the `Ctrl+R` shortcut.
If an index is set in the dataframe, table (G) can be used to select a subset of indices to plotted. When the chart is created, data are grouped by index, and every group is considered a different time series to plot. This feature was included in order to plot time series which have been extracted by longitudinal datasets. The documentation of the *Time series extraction* operation provides more details about this.

B.1.6 The Flow panel

The last available panel provides an alternative approach to dataset transformation: the same operations that could be applied singularly from both the *Attribute* and *View panel* can be chained together to form a pipeline where the output of a node becomes the input of its successors. Fig. B.9 displays a pipeline composed of 5 steps.
Steps can be added to the pipeline canvas (P) by dragging them from the list of operations on the left side of the window (A). Every step is represented graphically with a node that has some *knobs* (K) on the left an right side. Nodes can be connected by clicking a *knob* on the source node and dragging the interactive edge up to the

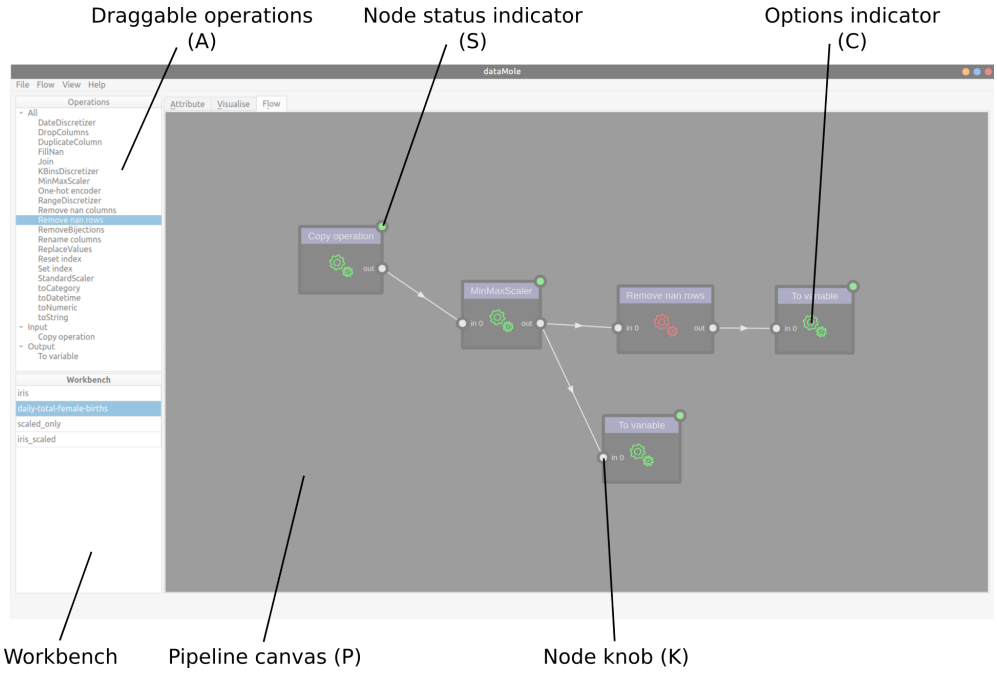


Figure B.9: A simple pipeline defined in the *Flow panel*

target node, where it must be dropped. Most operations require some parameters before the pipeline can be executed. In order to do this every operation has a *widget editor* that is used to configure them. For instance the widget shown in Fig. B.10 is the editor widget for the min-max scaler operation. These editors can be opened by double clicking the operation nodes in the pipeline, and when options are set they can be confirmed with the *"Ok"* button. At this point the operation is configured an its *options indicator* (C) turns green.

The pipeline canvas can be zoomed and moved around using the mouse wheel. Pressing the `F` key while the canvas is focused fits the view to its content.

When the pipeline has been configured it can be executed by clicking `Flow > Execute`. Once started, a *status indicator* appears above every node (S). It is grey for nodes that must still be executed, green for completed nodes, yellow for running nodes and red for nodes that failed with an error. This status can be reset by clicking `Flow > "Reset status"`.

Pipelines can also be imported and exported using the respective entries in the menu bar: `Flow > "Load"` and `Flow > "Save"`.

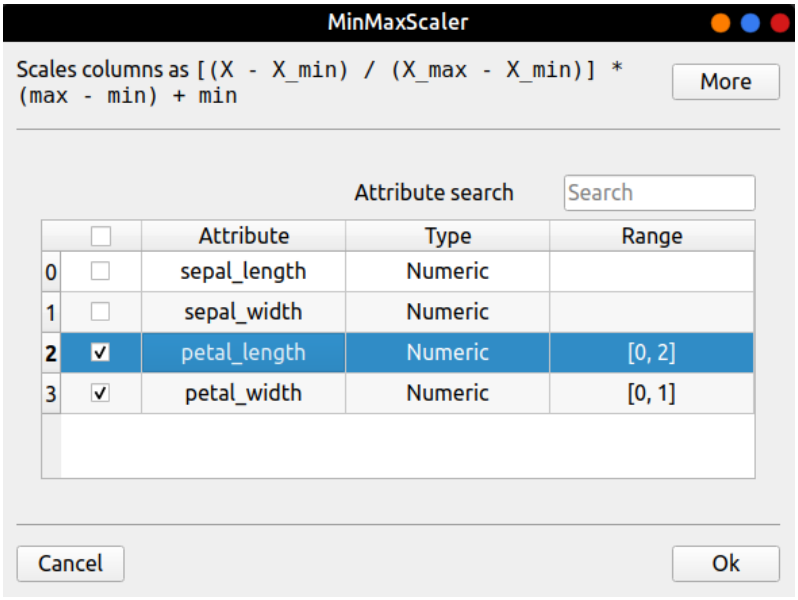


Figure B.10: The *editor widget* for the operation used to scale columns

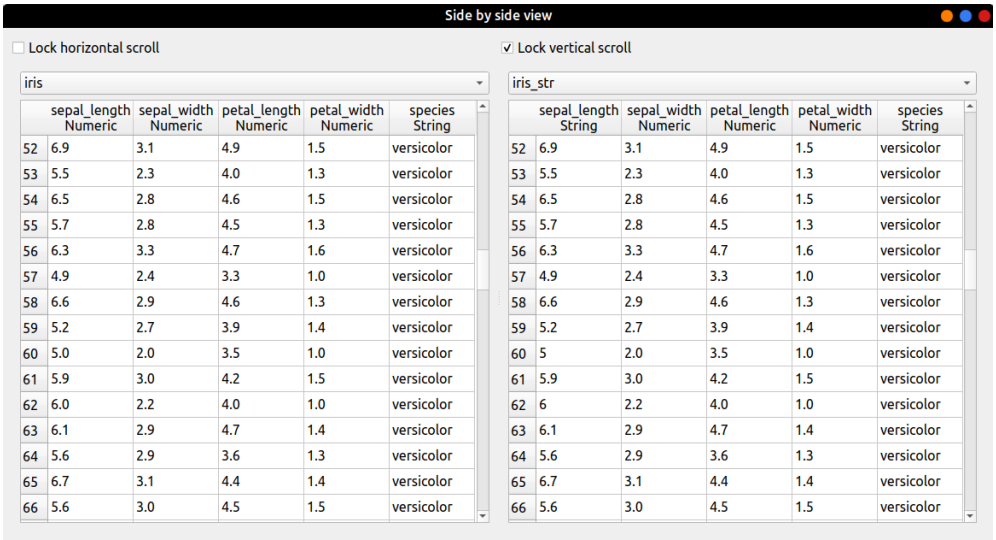


Figure B.11: Comparison of two dataframes side by side

B.1.7 Other features

B.1.7.1 Dataframe visualisation

Menu entry `View > "Compare dataframes"` opens a window where two dataframes can be selected and compared side by side, like in Fig. B.11. It is also possible to visualise single dataframes in a similar table, by clicking button "View as dataframe" in the *Attribute panel*.
These views are read-only, thus datasets can not be edited from within DataMole.

B.1.7.2 Logging facilities

Whenever a pipeline is executed from the *Flow panel* DataMole writes an execution report inside the `logs/graph` folder. Every log file is named with the timestamp of when it was created and includes information on the configuration of every operation (like user supplied options) and eventual parameters computed during execution. Also operations applied singularly from the *Attribute panel* are logged in the same way inside the `logs/ops` folder. In this case every program session initialises a new log file. Additionally debug information as well as critical errors are logged inside `logs/app`. These logs can be useful to debug software crashes and unexpected behaviours.
The log directory can be opened using the predefined window manager from menu `Help > "Open log directory"`.
Finally, option `Help > "Delete old logs"` clean the log directory removing all but the 5 most recent files.

Bibliography

References

[1] J. Banks et al. *English Longitudinal Study of Ageing: Waves 0-8, 1998-2017*. 2019. URL: http://doi.org/10.5255/UKDA-SN-5050-16 (visited on 09/09/2020) (cit. on p. 4).

[2] S. Bozena. *DataPreparator*. 2013. URL: http://www.datapreparator.com (visited on 09/09/2020) (cit. on p. 7).

[6] *ELSA: Study Documentation*. URL: https://www.elsa-project.ac.uk/study-documentation (visited on 09/09/2020) (cit. on p. 4).

[7] R. A. Fisher. *THE USE OF MULTIPLE MEASUREMENTS IN TAXONOMIC PROBLEMS*. Vol. 7. 2. 1936, pp. 179–188. DOI: 10.1111/j.1469-1809.1936.tb02137.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1469-1809.1936.tb02137.x (cit. on p. 11).

[8] E. Frank, Mark A. Hall, and Ian H. Witten. *The WEKA Workbench*. 4th edition. Morgan Kaufmann, 2016. eprint: https://www.cs.waikato.ac.nz/ml/weka/Witten_et_al_2016_appendix.pdf (cit. on p. 7).

[9] *Health and Retirement Study, (RAND HRS Longitudinal File 2016 (V2)) public use dataset*. Produced and distributed by the University of Michigan with funding from the National Institute on Aging (grant number NIA U01AG009740). 2020 (cit. on p. 4).

[10] *Modules - Python 3.8.5 documentation*. URL: https://docs.python.org/3/tutorial/modules.html (visited on 09/09/2020) (cit. on p. 30).

[13] *Qt Concurrent*. URL: https://doc.qt.io/qt-5/qtconcurrent-index.html (visited on 09/09/2020) (cit. on p. 44).

[15] *Qt Documentation: Graphics View Framework*. URL: https://doc.qt.io/qt-5/graphicsview.html (visited on 09/09/2020) (cit. on p. 38).

[16] *Qt Documentation: Model/View Programming*. URL: https://doc.qt.io/qt-5/model-view-programming.html (visited on 09/09/2020) (cit. on p. 29).

[17] *Qt Documentation: Signals & Slots*. URL: https://doc.qt.io/qt-5/signalsandslots.html (visited on 09/09/2020) (cit. on p. 29).

[18] *Qt Documentation: The Qt Resource System*. URL: https://doc.qt.io/qt-5/resources.html (visited on 09/09/2020) (cit. on p. 76).

[19] *Qt for Python: The official Python bindings for Qt*. URL: https://www.qt.io/qt-for-python (visited on 09/09/2020) (cit. on p. 28).

[20] *QThreadPool and QRunnable: Reusing Threads*. URL: https://doc.qt.io/qt-5/threads-technologies.html#qthreadpool-and-qrunnable-reusing-threads (visited on 09/09/2020) (cit. on p. 43).

[21] *Technical vision for Qt for Python - What lies ahead*. 2019. URL: https://www.qt.io/blog/2019/08/19/technical-vision-qt-python (visited on 09/09/2020) (cit. on p. 52).

[22] *Threading Basics: GUI Thread and Worker Thread*. URL: https://doc.qt.io/qt-5/thread-basics.html#gui-thread-and-worker-thread (visited on 09/09/2020) (cit. on p. 43).

[23] *What is PyQt?* URL: https://riverbankcomputing.com/software/pyqt/intro (visited on 09/09/2020) (cit. on p. 28).

Online resources

[3] *Daily total female births in California, 1959*. URL: https://www.kaggle.com/dougcresswell/daily-total-female-births-in-california-1959 (visited on 09/09/2020) (cit. on p. 15).

[4] *dataPreparation: Automated Data Preparation*. URL: https://cran.r-project.org/web/packages/dataPreparation (visited on 09/09/2020).

[5] *dsideb/nodegraph-pyqt*. URL: https://github.com/dsideb/nodegraph-pyqt (visited on 09/09/2020) (cit. on pp. 31, 38, 51).

[11] *Pipelines and composite estimators*. URL: https://scikit-learn.org/stable/modules/compose.html (visited on 09/09/2020) (cit. on p. 52).

[12] *pytest-qt*. URL: https://pypi.org/project/pytest-qt (visited on 09/09/2020) (cit. on p. 49).

[14] *Qt Documentation: Callout Example*. URL: https://doc.qt.io/qt-5/qtcharts-callout-example.html (visited on 09/09/2020) (cit. on p. 48).