UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di Laurea triennale in Ingegneria Informatica e dell'Automazione

"Machine Learning Algorithms for learning processes identification during educational robotics experiences"

"*Algoritmi di Machine Learning per l'identificazione dei processi di apprendimento durante esperienze di robotica educativa*"

Relatore: Chiar.mo

Prof. **David Scaradozzi**

Tesi di Laurea di:

Ludovico Svampa

Matricola 1068450

Correlatori:
Laura Screpanti
Lorenzo Cesaretti

A.A. **2021** / **2022**

Contents

Riassunto in lingua italiana

In questo elaborato viene descritta la struttura del prototipo “MLToolBox”, una toolbox ideata con lo scopo di supportare ricercatori e insegnanti nella raccolta e processamento dati derivanti da un esperimento di Robotica Educazionale (ER). Il lavoro svolto in questa tesi automatizza molti dei passaggi fondamentali che intercorrono dalla collezione dei *log file* all’addestramento dei metodi di *machine learning* selezionati dagli autori del progetto. Il software sviluppato è stato ideato in modo da poter essere facilmente utilizzato su dispositivi diversi garantendo la corretta riproducibilità degli esperimenti.

A partire dallo stato dell’arte su cui si basa lo sviluppo della MLToolBox, verrà descritta la struttura della stessa del software, le scelte di design e gli strumenti utilizzati. L’intera infrastruttura è costituita da quattro moduli che ne caratterizzano gli aspetti principali:

- `discor_module`: in questa unità è definita la classe `DiscordBot` che, sfruttando la libreria Python `Discord.py`, consente di inizializzare e gestire un bot in grado di collezionare localmente i *log file* prodotti dagli studenti. Tramite questo modulo, il sistema crea canali Discord appositi, associati ad ogni microcontrollore, da cui leggere e collezionare messaggi sotto forma di file di testo.
- `parser_module`: questa unità definisce la struttura di un vero e proprio parser in grado di manipolare il contenuto dei *log file* forniti in input. Sfruttando la libreria `Lark`, ogni stringa viene interpretata e codificata secondo una precisa logica (Parser Logic/Grammar) per compattare in forma numerica l’info in essa contenuta.
- `feature_matrix_module`: questa unità sfrutta le librerie `pandas` e `numpy` per processare una lista di *log file* ed estrarne la matrice delle caratteristiche. Quest’ultima è formattata rispettando le direttive di progetto dell’approccio Supervisionato [3] e costituisce l’input degli algoritmi di machine learning.

- `machine_Learning_module`: grazie alla libreria Python `ScikitLearn`, questa unità si occupa di gestire la fase di addestramento e testing dei quattro algoritmi di machine learning scelti dagli autori del progetto:
    - Support Vector Machine
    - Random Forest
    - K-Nearest Neighbour
    - Logistic Regression

Ogni modulo è stato progettato in modo da poter essere utilizzato in modo indipendente dagli altri, consentendo ad un utente esperto di utilizzare le sue funzionalità in differenti contesti. La struttura ampiamente modulare dell'intero sistema, ottenuta grazie al largo impiego di una programmazione orientata ad oggetti, consente a futuri sviluppatori di aggiornare il codice dei singoli moduli e di ampliare le funzionalità della toolbox.

MLToolBox dispone inoltre di una interfaccia utente che consente la gestione dell'intero sistema anche a programmatori meno esperti, poco abituati all'uso del terminale, mostrando a schermo i risultati delle varie fasi di processamento. Essa sfrutta funzionalità asincrone per l'esecuzione dei moduli, e si limita a visualizzare e salvare localmente solo le informazioni principali. L'utente può dunque utilizzare l'intero sistema tramite GUI in modo fluido e intuitivo, mantenendo gli stessi vantaggi dell'interazione da terminale.

L'obiettivo raggiunto con la realizzazione della MLToolBox è stato quello di fornire strumenti automatizzati e di supporto per effettuare esperimenti di ricerca legati all'ER. Il sistema è in grado di riassumere e gestire in pochi comandi la grande mole di processamento dati relativa ad ogni fase, facilitando e supportando il ricercatore nel proprio esperimento. La toolbox rappresenta un primo prototipo che può essere integrato, ampliato con nuovi moduli, e adattato ad essere utilizzato in settori di indagine simili a quello per cui è stato ideato. La MLToolBox permetterà di migliorare la raccolta dati e di raggiungere quindi nuove conclusioni riguardo la relazione tra le attività ER e il grado di apprendimento degli studenti.

Introduction

The main objective of the Educational Robotics (ER) is to help students explore powerful ideas and authentic learning in open-ended environments, where hardware and software tools allow them to explore and create solutions to a given task [1]. There is therefore a need to assess the skills acquired by the student during the course and to test his or her level of learning in order to certify them in a curriculum. The approach adopted in the evaluation methodology is qualitative, but it can be influenced by the observer's bias and external factors, so it is necessary to create an infrastructure that helps the teacher to manage and evaluate students more objectively. By exploiting Educational Data Mining (EDM) and Learning Analytics (LA) it is possible to identify and model students' learning and assess ER activities in a real classroom environment by automatically and systematically collect and analyse data and providing clear and understandable results to teachers [2].

Numerous problems arise in the design of such identification systems, like for example the nature of data to collect and the associated methods to gather such data, or the identification of features that balance the need for human understanding and model's implementation [3]. Inspired by the existing literature answering those questions, this thesis will illustrate the development of a ToolBox designed to analyse data gathered from the learning process in a classroom while learning with Educational Robotics to support the teacher during an ER course.

The 'SMART BLOCKS FOR TEACHING ROBOTICS' technology, patented by Università Politecnica delle Marche, describes an experimental system that automatically and in real time gathers and processes online data of students interacting with robotic toolkits during a ER lesson. In particular, the proposed system elaborates the collected data in order to evaluate the strategy that students followed during the exploration of an open ended ER activity. The proposed model has a high added value for students and educators, because it empowers teachers with a first evaluation of students' learning status allowing them to provide students with immediate feedback [4].

Until now, the developers of this patent have independently realised some of the components of the system and tested them in relevant and operational environments; the results of these first experimentations are available in [1], [8] and [3]. A further step in the development of the system is the automation of the data analysis and the creation of a framework that merges all the steps of the proposed system, allowing teachers to use the entire infrastructure more easily and efficiently. It also provides access to the documentation and example scripts to facilitate the understanding of the code and future modifications. The toolbox, implemented and discussed in this thesis, is named MLToolBox and represents a container for the system by storing the entire code and the dependencies needed to execute all the required functionality for the purpose of the experiment: thanks to it, it will be possible to transfer the entire infrastructure from one computer to another, guaranteeing the reproducibility of the entire experiment.

1. Related works

In the learning environments based on project-based pedagogy, such as Educational Robotics, students often take unpredictable paths for solving open-ended problems and it is difficult for the teacher to help them properly during their learning process, ensuring optimal results. Numerous studies aimed to develop a system able to support teachers and analyse the behaviour of students, looking for a link between their problem solving attitude and their performance in computer programming. Employing the use of machine learning and clustering methods, the researchers tried to highlight meaningful patterns in the collected data and to automate their systems for ever more in-depth results.

1.1. Jormanainen and E. Sutinen

The first to devise initial support for educators were Jormanainen and E. Sutinen in the 2000s with the Open Monitoring Environment (OME), that allows the teacher to monitor and model the learning process based on data emerging from the ongoing learning setting [5]. Thanks to their system, equipped with semi-automatic tools, it was possible to manage the large and complex amount of data extracted from the students' learning process; the information obtained was then used to test various data mining methods and the results were reviewed by the teachers.

The development of the OME was initially based on the collection of empirical data from an ER setting where 12 South African school children and 4 teachers were working in a robotics project. With this dataset, the authors of the project concentrated on analysing the performance and efficiency of various mining methods, testing them in different clustering environments.

Considering the J48 algorithm, best-first decision tree (BFTree) algorithm, and multi-layer perceptrons (MLP) the best for categorising student behaviour, Jormanainen and Sutinen implemented those data mining methods in their system: the

process “is open to the teacher and it allows him or her to control the functionality of an automatically created classifier in detail” [5].

The OME aims to filter and group the data extracted from the learning process through a data mining process, and to give teachers the opportunity to question the current results of the system. “Allowing the educator to experiment with the data mining tools and reflecting the changes immediately to the monitoring environment support creative thinking and suggest new explanations behind the current situation in the robotics classroom” [5].

1.2. P. Blikstein and C. Piech

While the work of Jormanainen and E. Sutinen aimed to support the teacher's intervention strategies, other researchers focused more on how students learn computer programming, using methods from machine learning to discover patterns in the data and trying to predict final exam grades [6]. In particular, P. Blikstein and C. Piech began with a set of exploratory experiments that used fully automated techniques to investigate how much students change their programming behaviour throughout all assignments in the course. They tried to map students’ learning process and trajectories and automatically identify productive and unproductive states within these trajectories [6].

Blikstein and Piech based their work on the earlier studies of Papert and Turkle who had noted how computer programming can itself be regarded as a record of students’ cognitive processes [6]. They developed an automated system to collect more than 154,000 code snapshots captured from the assignments of 370 students, during an introductory programming methodology course. After collecting this data, they performed numerous tests to track the students' progress using machine learning techniques. Their experiment was structured in two parts: on one side they wanted to trace the trajectory of an individual student while performing an exercise; on the other

side they looked for possible patterns across several assignments and tried to correlate those patterns with students' assignment and exam grades [6].

Assuming that programming is not just generating code, but a deeper process of thinking, decomposing, and solving problems, Blikstein and Piech attempted to measure students' behaviours in the following ways. Beginning with the comparison of two consecutive code snapshots, they determined how much change has taken place, from one snapshot to the next, by measuring the number of characters or lines of code that students have added, removed, or modified. This comparison allowed them to calculate a set of six measures: number of **lines added**, **lines deleted**, **lines modified**, **characters added**, **characters removed**, and **characters modified** [6]. They defined this set as the "code update differential".

The next step was focusing on the code update pattern (size and frequency of code updates), assuming that small, frequent code changes could represent episodes of **tinkering**, whereas larger, less frequent code changes could represent episodes of **planning** [6]. Thus, Blikstein and Piech generated code update curves for each student; they used them to calculate a general score for students' pattern changes during the classes, and find correlations between the amount of change and course grades.

The results of their research were just a first step in the direction of developing measures for programming patterns such as tinkering and planning, or how much students change their patterns. This seemed to agree with the early literature in this nascent field, testifying a relationship between the ways in which students update their code and their programming experience [6].

1.3. P. Y. Chao

Subsequent studies “revealed that visual programming environments could enhance novice programmers' engagement in programming tasks and help them demonstrate programming skills and problem solving strategies during the course of creating digital artefacts or solving programming problems” [7]. For this reason, P. Y. Chao decided to develop a visual problem solving environment for learning programming, which would help him to collect data for his experiments. The proposed system provided students to represent design strategies and evaluate them with immediate feedback.

The "log-data" collected during the experiment represents participants' practice, strategies and performance in computational problem-solving activities. The author of the project decided to synthesise all the information with 5 indicators: **Sequence**, **Selection**, **Simple iteration**, **Nested iteration** and **Testing**. After that, Chao performed a cluster analysis on students’ frequencies of the five indicators: his purpose was to highlight possible patterns of computational practice for solving computational problems [7]. The k-mean algorithm, supported by Ward's minimum variance method, suggested a four-cluster solution showing clearest distinctions among and providing more meaningful explanations for the different patterns of computational practices [7]. Students’ behaviour were classified as:

- **Sequent approach:** the participants in this cluster tended to compose solutions with a relatively linear progression of computational practice, such as sequence or simple iteration, but tended to exert less effort to use more advanced practices, such as selection or nested iteration.
- **Selective approach:** the participants in this cluster implemented different action plans corresponding to different conditions; were likely to produce a more flexible and efficient solution.
- **Repetitious approach:** the participants in this cluster mainly applied a simple repetitious approach to computational practices while performing the computational problem-solving activity.

- **Trial approach**: the participants in this cluster tended to combine various computational practices in the problem-solving activity and frequently employed a trial approach to testing the results of generated instructions; they seemed to try the application more frequently and evaluate the consequence of their implementation.

1.4. Scaradozzi, Screpanti and Cesaretti

ER activities allow students to observe and improve their robot, keeping them motivated to learn. According to previous studies, “data mining techniques applied to data collected during ER projects could allow the identification of the students’ programming skills levels or the levels of improvement time by time” [8]. The system proposed by the Politecnica delle Marche University patent “takes the existing works further through the utilisation of educational robots and machine learning techniques to allow a rich and a more in-depth evaluation of the students’ results and to exploit the potential of automated assessment for the classroom monitoring and modelling, and for supporting teacher’s evaluation” [3].

D. Scaradozzi, L. Cesaretti and L. Screpanti carried out their first experiment from March 2018 to March 2019, providing 7 Italian secondary schools (197 students) with a robotic toolkit like the Lego Mindstorms EV3 kit. The participants used the Mindstorms EV3 IDE to program their robot, choosing from a range of blocks that visually represent a set of functions. The authors of the project wanted to collect all the coding interactions the students would have performed on the robot. So they modified the Lego kit allowing their system to gather all the data as log files that will be analysed to find evidence of the students' learning process.

In this way, teachers can be informed about students' problem-solving approaches and about the prediction of their success in a given exercise; moreover, at a higher level, it’s possible to keep track of students' progress over time.

Authors explored two feature extraction approaches to analyse the log files and to prepare data for the machine learning algorithms [3]:

- **Supervised approach:** each test produced by the student is summarised through indicators established by the researchers. These indicators highlight the blocks of codes chosen by the student and the structural differences between two consecutive code sequences. Then, the **mean value** and the **standard deviation** are calculated for each indicator producing a bidimensional vector for each log file.

- **Mixed approach:** all the programming sequences are analysed by a cluster algorithm (i. e. K-means supported by the Elbow method) to find a perfect number of clusters, highlighting possible patterns**.** Then the system analysed the sequences in each log file to identify the presence of sequences belonging to clusters. “The result of this analysis is the percentage of each cluster in the programming activity of the students' groups, so that a vector that is composed of the percentages of sequences that belong to each cluster is created for each team” [3].

The result of both methods is a multidimensional **feature matrix** that will be given as input to four machine learning algorithms namely **Logistic Regression (LR)**, **Support Vector Machine (SVM)**, **K-Nearest Neighbors (KNN)**, and **Random Forest (RF)**, in order to predict students’ performances.

By analysing the code sequences associated with each cluster in more detail, the project authors (aided by teachers) were able to associate the obtained clusters with three most common student behaviours:

- "**Math/planning**", low number of trials and none or small changes to the programming sequence.

- "**Tinkering with refinement**", heuristic approach to decide the parameters of the robot, and repeatedly refined them after analysing its feedback.

- "**Tinkering with significantly high modifications**", heuristic approach to decide the parameters of the robot, a high number of large modifications after analysing its feedback.

The last part of the experiment focused on predicting students' results by applying previously trained machine learning algorithms. D. Scaradozzi, L. Cesaretti and L. Screpanti wanted to compare the performances of the Supervised and Mixed approaches to explore two different strategies of feature selection, as already proposed by other researchers in similar tasks [3]. In fact, the **Supervised approach** produces a feature matrix of information deemed relevant by human experts; the **Mixed approach** is free from human bias while selecting features.

"Results showed that the mixed method had a better performance than the supervised method, and the SVM techniques, in combination with the features calculated by the K-Means algorithm, outperformed the other machine learning algorithms" [3]. From a pedagogical point of view, the authors system can help teachers in the identification of those students that are at risk of failure providing them with realtime support.

What emerged from this first attempt is intended to be confirmed and/or further investigated by subsequent experiments on a larger scale: the research team therefore needs an infrastructure that is as automated as possible, facilitating the execution of the various steps and making the first experiment replicable. Numerous thesis and PhD students have contributed with their own projects to extend and improve the system proposed by the Politecnica delle Marche university's patent, but they created partial and non-uniform systems. The result is the development of the ToolBox that is the main subject of this paper: a single infrastructure equipped with necessary functions to help and facilitate the authors of the project in future replications of the experiment.

2. Tools and methods

This chapter will delve into the structure of the MLToolBox and the design choices made during its development. It represents an initial prototype system that automates the various steps from data collection to the training of machine learning methods.

This system was also designed to facilitate as much as possible future integrations and modifications to the entire apparatus.

2.1. Experiment structure

According to the structure of the first experiment performed by Scaradozzi, Screpanti and Cesaretti [3], the basic steps to be replicated are:

1. Provide the student groups with a robot that can be programmed to perform a specific task. The code can be developed by students thanks to a visual programming environment. Hacking the kit it will be able to generate log-files.
2. The code within the log-files must conform to the rules of a precise grammar/syntax, established by the authors of the project.
3. Log files must be collected and saved for easy reading and analysis.
4. The information contained in the code sequences must be processed according to one of the two approaches proposed in the project (i.e. Supervised approach and Mixed approach [3]). Thus, the system needs to produce the feature matrices to feed the algorithms.
5. Four machine learning algorithms (i.e. SVM, RF, LR and K-NN) must be trained using the feature matrices.
6. Save and reload trained algorithms to run predictions or perform tests.

A good system should automate most of these steps (from point 2 to 6) and display partial information during data processing. In this way, users, which could be researchers and also teachers, can check and be reassured about the correct execution of the infrastructure and monitor the students' work in real time as well. In addition, it must be possible to perform some of these steps individually, allowing the user to focus more on a precise phase of the experiment.

2.2. Previous projects

The development of this MLToolBox is based on the three previous projects, which bounded some of the resources used and some structural aspects:

- **Study and Development of "Talking" Educational Tools** [9]**.** In this work, the author aimed at modifying the software of the M5StickC, the microcontroller inside the robot that will be used in future experiments. The author created a custom library for the UIFlow [21] visual programming app and enabled the log file generation starting from the robot which is used by the students during their ER activities. This project **defined the grammar/syntax** to codify the string of code produced by the students: this e grammar/syntax will be used to interpret the information stored in the log files.

- **Educational Robotic Tools for the Identification and Modelling of Learning** [10]**.** This project focuses on the collection of log files created by [9]. The author developed a communication system between the M5StickC and a **Discord** [22] server based on a protocol that sends the log files made of sequences of code (produced by the students) as a message on a dedicated Discord server. By collecting all the messages from the same microcontroller, authors can recreate the log file corresponding to that bot's use and save it.

- **How students solve problems during Educational Robotics activities: identification and real-time measurement of problem-solving patterns** [11]. In this work the author developed the infrastructure used in the first

experiment. He defined which information to extract from the log files, structuring the **indicators** needed to create the feature matrix in the Supervised approach. **Metrics** for comparing the various code sequences were also chosen in his system.

Although an existing working system that gathers and analyses data from students' interactions with a robotic toolkit has been already developed by [inserisci references di Toward a definition of educational robotics, frontiers, IEEE RAM], as explained in 1.4, further developments were needed.

According to the directives of the first experiment, the infrastructure was designed to perform data analysis from *Exercises A* and *Exercises B* [11] as efficiently as possible. The system expects a specific type of input and is therefore difficult to use in future experiments with different tools and dynamics. Furthermore, the structure of the code itself precludes the intervention of new programmers to introduce new features and/or modifications. Lastly, the existing algorithm was developed to read log file with a different grammar/syntax, so the toolbox was given the capability to conform to the new syntax, that was described by [9]

The ToolBox, subject of this article, offers a system that can standardise previous projects and ensure in-depth analysis of information. The structure of the code itself has been developed to ensure a certain level of abstraction from the **type** of experiment it should support: this makes it easy to expand and improve the system by adding new features and tools.

2.3. Project dependencies

This section will deal with the programming language and libraries used for the development of the ToolBox, briefly describing them and explaining their choice. The system is based on the following dependencies:

- **Python**: one of the most widely used programming languages in the field of Data science and Machine learning.
- **Numpy, Pandas** and **Matplotlib:** Python libraries designed for data analysis and manipulation. They provide lots of functionalities and *structures* to easily perform mathematical operations and plot information.
- **Lark:** Python parsing toolkit able to generate parsers for the provided grammar.
- **Discord (Pydiscord):** Python library that provides the ability to communicate with the Discord API, a VoIP and messaging application.
- **Scikit-learn:** a FOSS Python library for Machine learning, featuring various classification, regression and clustering algorithms including support-vector machines, random forests, gradient boosting, k-means, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy [12].
- **Colab:** it is a Google service based on an open source project called *Jupiter* that allows anybody to write and execute arbitrary Python code through the browser, and is especially well suited to machine learning, data analysis and education [13].
- **GTK:** graphic toolkit designed for the creation of user interfaces.
- **Flatpak:** a packaging technology designed to be secure and reproducible across the systems.

2.3.1. Python

Figure 1: Python logo

Python is a high-level interpreted programming language that supports various paradigms such as structured, object-oriented and functional programming. It's dynamically typed and uses a garbage collector. In Computer Science, garbage collection (GC) is a form of automatic memory management. The garbage collector attempts to reclaim memory which was allocated by the program, but is no longer referenced—also called garbage [14].

Python language is widely used in the fields of Machine Learning and Data Science due to the large number of constantly updated and supported libraries. In addition, the support for **object-oriented programming** helps to develop and use complex infrastructures more easily.

It also has a **packaging feature** that has been exploited to make the entire system portable.

Python packaging

Python's packaging has not been standardised and there are different solutions based on the same principle: as long as Python scripts can find their dependencies, they will work. However, the Python package `setuptools` provides a "default" way to do this.

The packaging process takes place in a specially created Python environment: in this way the security is improved and it is easier to detect missing dependencies. To package a project, it needs to be modelled in a specific way to allow `setuptools` to

gather all the information it needs. This is an example of typical Python package structure:

```
packaging_tutorial/
├── LICENSE
├── pyproject.toml
├── README.md
├── setup.cfg
├── src/
│   └── example_package/
│       ├── __init__.py
│       └── example.py
└── tests/ [15]
```

The `setuptools` build script will look for some specific files in the project folder (some are optional and others are instead mandatory):

- `LICENSE`: this file is optional and contains the licensing information of the packaged code.
- `README.md`: this file is optional and contains some readable information about the package. It usually provides details about *build dependencies*, *runtime dependencies*, the *packaging command* and more.
- `pyproject.toml`: this file is **mandatory** and has to be in the root of the project. It is written using the `toml` format and contains, among other things, the compilation dependencies that will be installed in the virtual compilation environment and the compilation backend that will handle the packaging.

  The `setuptools.build_meta` backend is always available, but it is also possible to provide custom scripts to manage the process.
- `setup.cfg` - `setup.py`: It is mandatory to have at least one of these two files in the project root. They provide all the information that is required to package the program, in particular which files are needed to be included.

`setup.cfg` is a static file, (it is just parsed to get the required information); `setup.py` is an actual Python script that is executed, so it can perform dynamic operations, like doing pre-packaging operations.

A Python project can provide multiple Python packages: using the setup files it is possible to configure where `setuptools` should look for packages (usually the src folder inside the project root). Every package must have an `__init__.py` file (even an empty one) for being detected by `setuptools` as a Python package.

Python async feature

Python supports the creation of **asynchronous functions** by using the `async/await` keywords.

An *asynchronous function* is a special kind of function where its execution can be eventually paused in case of external input/output operations, returning the control to the system scheduler and allowing concurrent executions. By calling this kind of function, its content is not executed immediately and a Python coroutine is generated. To be executed, its content must be scheduled in an "executor" that encapsulates the coroutine in a structure keeping track of executor-specific data.

Python provides asynchronous features through a library called `asyncio` and its executor offer two coroutine wrappers:

- `Future` class: a low-level wrapper for asynchronous function execution
- `Task` class: a high-level wrapper for asynchronous function execution

2.3.2. Lark

Figure 2: Lark logo

In the first experiment, [11] was equipped with a complex block of code that manually simulated the parsing process. Thanks to Lark, it is possible to instantiate and exploit all the features of a real parser.

It is a Python parsing toolkit able to generate parsers for the **provided grammar** in Extended Backus-Naur Form (EBNF) metasyntax notation, automatically generating its Abstract Syntax Tree (AST). Lark supports a wide range of input grammars, providing both an Earley parser and a Look Ahead Left-to-right Rightmost (LALR) parser.

Based on the *syntax* defined by [9] and [10], Lark is able to read and encode any text file recognised as valid input. The use of a parser makes it possible to collect information about the structure of code sequences in log files, and display/save them.

In the following sections, the parsing module that mainly benefits from the use of this resource will be described.

2.3.3. Discord/Pydiscord

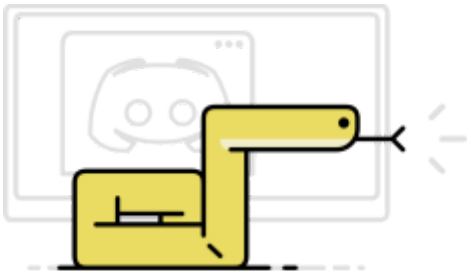


Figure 3: Discord.py logo

According to [10], the M5 Stick microcontroller was provided with a communication protocol with Discord. Her "log file collection system" is only partially automated: the messages sent by the robot on the Discord chat must be saved manually by the researcher in a special text file.

Pydiscord is a Python library providing the ability to communicate with the API of Discord. It was used to fully automate the log file collection process. Thanks to Pydiscord, it was possible to implement a bot in the `discord_module` that can read specific chat messages and collect them locally as text files.

2.3.4. ScikitLearn

Figure 4: ScikitLearn logo

The ScikitLearn library was chosen by the project authors for the Machine learning part of the project: it is one of the main libraries used in this field, so it is enormously supported. ScikitLearn presents numerous tools for the instantiation,

training and utilisation of the main algorithms used to classify and predict students' behaviour; it can therefore be easily integrated into a complex data analysis system.

2.3.5. Google Colaboratory

Figure 5: Google Colab. logo

This Google service was chosen to provide detailed documentation of the code contained in the Toolbox. Each module has an in-depth link to a specific notebook, enabling new researchers and designers to easily read and understand the algorithm and view example scripts.

The code contained within the Colab files can be executed online without the need to install dependencies locally. In addition, the example scripts serve to illustrate the terminal commands that can be used by the researcher.

2.3.6. Gtk

Figure 6: Gtk logo

GTK (GIMP ToolKit) is a FOSS (Free and Open Source) cross platform graphic toolkit for the creation of user interfaces, licences under the terms of LGPL. It offers various backends to support multiple platforms, including Linux (both X11 and Wayland), Windows, OS X and Web. It has evolved into multiple revisions as Gtk2 (deprecated), Gtk3 (in maintenance mode) and Gtk4 (the latest and current version).

GTK is a combination of various libraries:

- **GDK** (GTK Drawing Kit): It is a low level library providing an unified interface over multiple platforms. GDK lies between the display server and the GTK library, handling basic rendering such as drawing primitives, raster graphics (bitmaps), cursors, fonts, as well as window events and drag-and-drop functionality [16].
- **GKS** (GTK Scene graph Kit): It provides a rendering API, to draw graphical control elements (widgets) [17], and a *scene graph* API to perform *scene aware optimization*.
- **GTK Introspector**: It is a development tool that allows to visualise the scene tree of a GTK application and expose all the widgets within it. It is also possible to see the properties of each widget and live-edit them.

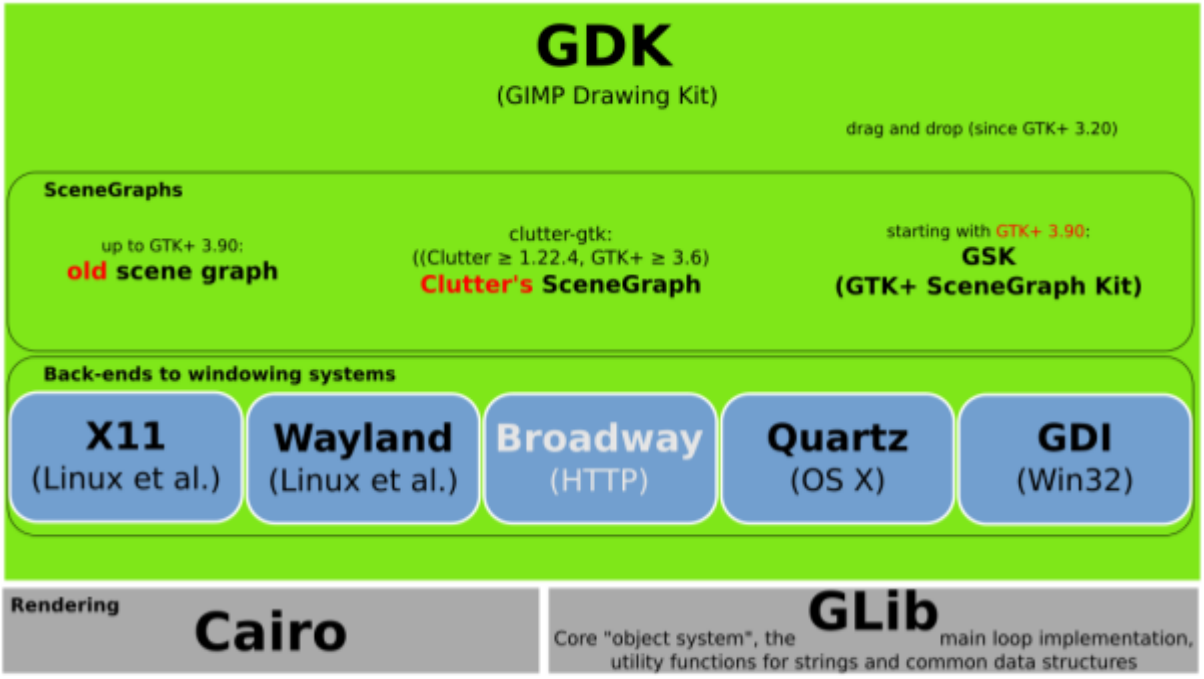


Figure 7: Gtk library structure

These libraries offer various widgets to suit design needs and each of them is a combination of three components:

- **Constructors** to define the instantiation procedure.
- **Methods** to set the functions available for an object instance.
- **Properties** to define the functional features.

Furthermore it is possible to alter the outlook of the widgets by applying custom **CSS** rules. CSS is a descriptive language designed for web development and is used in many fields that require deep customisation of visual properties by the user..

With the latest release of GTK, **Libadwaita** library can be used: it improves GTK by introducing new widgets and styles with the aim of providing a uniform appearance across applications without limiting user design choices.

Gtk offers three ways to develop and deploy an interface:

1. Write the interface using code only.
2. Write the interface using GtkBuilder UI definitions, an xml-based descriptive representation of the UI that will be parsed by GtkBuilder to generate the actual widgets and tree structure.
3. Write the interface using GtkTemplate, a class that can use GtkBuilder's UI definitions to extend the widgets with new methods and properties, allowing custom code to be tightly integrated with the descriptive UI representation.

Python bindings are officially supported with a library called **PyGObject**.

2.3.7. Flatpak

Figure 8: Flatpak logo

Flatpak is a packaging solution for Linux that can work together with other packaging systems. It is similar to a mini-root file system and contains everything it needs to be functional.

A Flatpak package is installed at user level, so it does not require system privileges. The user may, however, allow applications running within it to access system services (e.g. network and display protocols). Flatpak packages may require access to a Runtime and an Sdk (collections of dependencies shared by all Flatpak

applications): the first is used at runtime (i.e. runtime dependencies) and the second during the build process (i.e. build dependencies). These dependencies avoid conflicts with other applications because they are not shared with the system and are updated separately. When a Flatpak package requires a missing runtime, it is installed automatically.

Flatpak packages are built using a tool called `flatpak-builder` and an application description file containing all the information needed to create the package. The most important ones are:

- `app-id`: the application id, formatted using the reverse domain name notation (e.g. “org.example.App”).
- `runtime`: the runtime required by the application.
- `runtime-version`: The runtime version.
- `sdk`: The sdk required for building the package.
- `command`: The main application launched in the Flatpak package.
- `finish-args`: This is a list of arguments used during the “finishing” stage of the packaging to set the application’s sandbox permissions (e.g. access to network, devices, display sockets etc.).
- `cleanup`: This is a list of files or folders that can be used to remove files produced by the build process that are not wanted as part of the application, such as headers or developer documentation [18].
- `modules`: This is a list of dependencies the package requires to work. They can be retrieved from various sources, like local files, archives, gits and many build toolchains are supported like **meson**, cmake and autotools.

2.4. Project structure

This section will delve into the structure of the MLToolBox by describing its various modules and reporting significant portions of the code.

The system consists of four modules that implement the functionality needed to carry out the experiment, and a graphical interface that simplifies the use of the software. A modular structure was chosen for the following reasons:

- It is easier to make and support changes over time.
- it is easier to expand/improve the system by integrating new modules.
- It is also possible to use the functionality of a single module in other research contexts.

The structure of the project is displayed as follow:

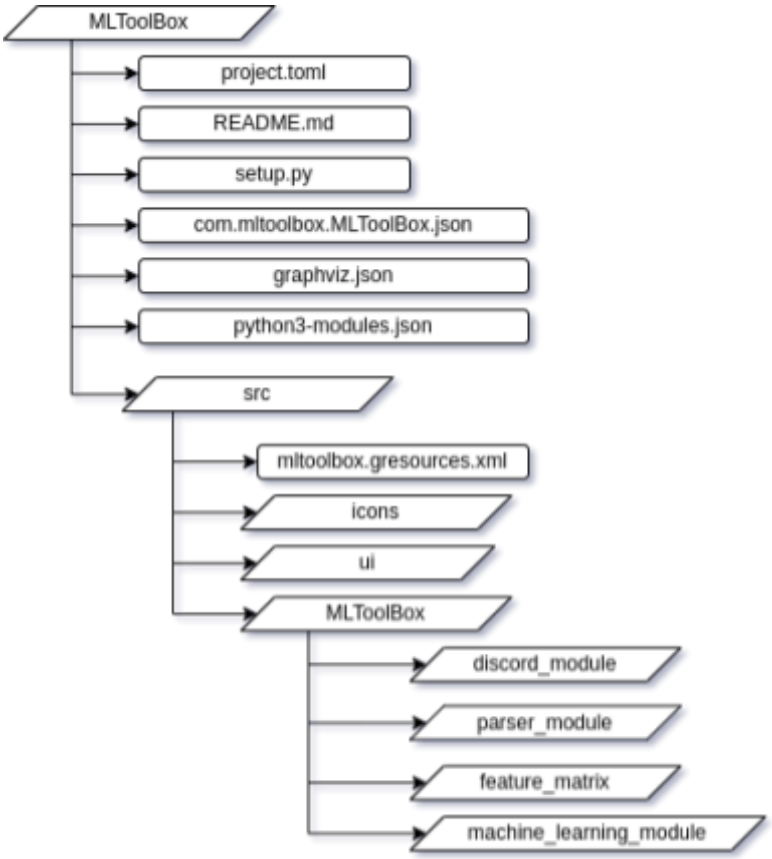


Figure 9: MLToolBox project structure

2.4.1. `discord_module`

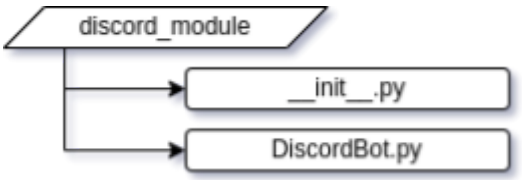


Figure 10: discord_module structure

This module automates the process of collecting log files. By implementing a bot in the Discord channel where the M5 Stick microcontrollers communicate, messages are read and saved within a text file, recreating a log file for each student's robot. It requires the following Python libraries:

- `os`: for os functionalities such as collecting/using files paths.
- `discord`: to develop the Discord bot.
- `Thread from threading`: to manage threads.
- `Enum`: to create an enumerator.
- `Asyncio`: to manage coroutine (async. functions)

and the entire code can be found in the `DiscordBot.py` file.

The logic behind the implementation of this module is based on two factors:

1. The difference between synchronous and asynchronous functions (previously discussed).
2. The correlation between **threads**, **loops**, **tasks** and **coroutines**.

When the system is started (either from the graphical user interface or from the terminal), it runs on the main thread. The use of the Discord bot implies the

scheduling of a **task** in a **loop** done by the **thread** and the `Discord.py` library offers two possible strategies:

- `Client.run()`, this is a **synchronous function** that runs the bot from the main thread. This causes the main programme to block until the "task-bot" is terminated. This implementation was immediately discarded in favour of the second one.
- `Client.start()`, this is an **asynchronous function** that uses a secondary thread. The Discord bot can be launched without blocking the execution of the main program.

The module defines the class `DiscordBot`, an extension of `discord.Client` class. The `discord.Client` class provides an already initialised **event loop** (of type `asyncio.Loop`) that is used to run the coroutines generated by Discord's asynchronous functions. Initialising a DiscordBot object automatically creates a **secondary thread** able to access the Discord loop.

```
def init(self, status_callback = None, terminal_launched=True):
    super().init()
    self.status_callback = status_callback

    # secondary thread
    self.thread = Thread(target=self.bot_thread)
    self.thread.start()
```

The command `.launch()` starts the bot: it sets a bot-**task** which is scheduled in the **loop**.

```
def launch(self, bot_name: str, channel_number: int,save_folder_path: str):

   self.change_status(BotStatus.LAUNCHING)
   self.bot_name = bot_name
   self.save_folder_path = save_folder_path
```

```
    self.channel_number = channel_number

  # Encapsulating a coroutine in a task provides additional functionality,
such as 'cancel'
  self.bot_task = self.loop.create_task(self.start(self.TOKEN,reconnect=True))

  # The task is added to the loop using a special  function `threadsafe`
  asyncio.run_coroutine_threadsafe(self.bot_task_runtime(), self.loop)
```

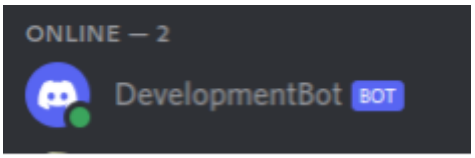


Figure 11: DiscordBot is online after the `.launch()` command

It is also possible to define a callback function that will be called whenever the bot changes state, thanks to it an event (e.g. generating changes to the user interface) can be reacted to. There are four possible states the bot can be in, defined in the `BotStatus` enumerator (i.e. stopped, stopping, launching, running).

```
class BotStatus(Enum):
    STOPPED = 0
    STOPPING = 1
    LAUNCHING = 2
    RUNNING = 3
```

Once started, the bot creates the channels requested by the user and will actively listen to the messages sent in the chats.

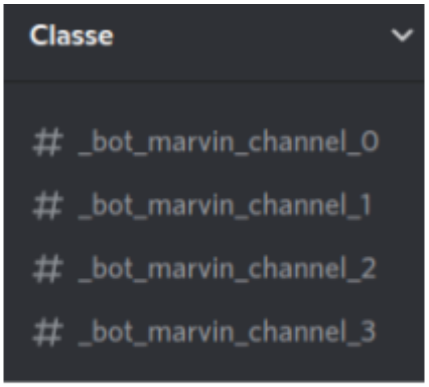


Figure 12: examples of channels initialised by the DIscordBot

If a message starting with `#` is sent in a channel chat (Figure 13), the bot collects it locally in a text file with the same name as the channel. It will also write in the chat that it has recorded the message.

```
async def on_message(self,message):
  if message.author == self.user:
    return
  if not message.channel.name.startswith(self.channel_prefix):
    return

  if message.content.startswith("#"):
    print("Message detected")
    file_path = os.path.join(self.save_folder_path,message.channel.name)
    with open(file_path+".txt", "a") as logfile:
        logfile.write("\n"+message.content)
        await message.channel.send("Message collected")
   else:
    print(message)
```

Figure 13: Example of message read and collected by the DiscordBot

The bot remains active and performs its functions until it is terminated: using the `.terminate()` command the **task**-bot is cancelled, but the loop and the secondary thread are still active. This alters the status of the bot and deletes previously created channels. The user can use this method to launch a new bot with new parameters in the `.launch()` function.

The **secondary thread** remains active until the loop associated with it is blocked using the command `asyncio.Loop.stop()`. When this function is called, it causes an exception on the **event loop** that will be caught by the thread. At this point, the loop deinitialization procedures begin and the secondary thread is rejoined. Using the `.deinit()` command starts all the functions to shut down the bot and terminate the thread it is running on all at once.

```
    def deinit(self):
        if not self.loop.is_closed():
            self.loop.call_soon_threadsafe(self.loop.stop)
            self.thread.join()
```

2.4.2. `parser_module`

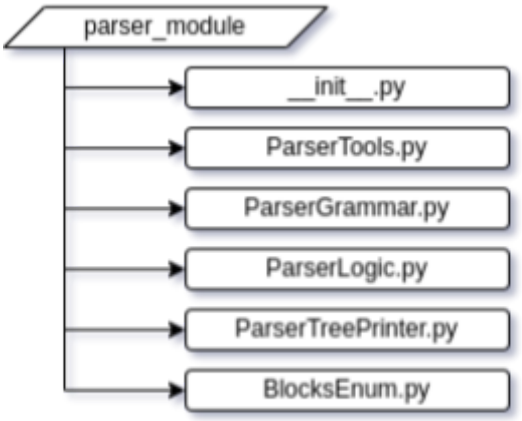


Figure 14: parser_module structure

This **module** deals with the analysis of *log files* (**i. e.** text files taken as input) and the numerical coding of their contents.

Each *log file* contains the code sequences produced by the student on each test; they are collected as **strings** and cannot be immediately used to extract the **feature matrix**. It follows that each sequence must first be processed and converted into a **numerical sequence**, which will then be handled by the `features_matrix_module`. The encoding of the strings is implemented through the development of a dedicated **parser**, designed to handle the *syntax* proposed in [9] and [10].

The proposed **module** requires the following Python libraries:

- `enum`: to create an enumerator.
- `lark`: to develop the **parser** which extracts and encodes the **test sequences** from the *LogFile* in text format.
- `pathlib`: for loading a text document (*input* expected by the system).
- `graphviz`: to display the results of the **parser**.

The entire code was segmented into the following Python files:

- `BlocksEnum.py`
- `parser_grammar.py`
- `ParserLogic.py`
- `ParserTreePrinter.py`
- `ParserTools.py`

each of them implements a specific module **component** and allows targeted modifications without compromising the functionality of the entire structure.

BlocksEnum.py

In this file, the class BlocksEnum has been defined: it is used to keep track of the numerical values associated with the *function-blocks* used by the students.

Here are encoded all the 'blocks' that the **parser** can process: their value must be unique and must be agreed with the authors of the project. For example:

```
LED_ON = 10001
LED_OFF = 10002
SUM = 10003
SUBTRACTION = 10004
MULTIPLICATION = 10005
DIVISION = 10006

[...]

SEQUENCE_END = 99999
```

Thanks to this enumerator, a test string is univocally converted into a **numeric sequence**.

ParserGrammar.py

The ***parser's* grammar** defines the *syntax* of the accepted *input language*! In this file, the **lark object** `parser_grammar` has been initialised.

Using the lark library, it is possible to define the **phrase structure** that the user wants to process: this determines what the *parser* will recognise as valid *input* or not.

In the code it is defined:

- `start` as an object consisting of one or more sequences. It represents the "grammar transposition" of the text document taken as *input* by the *Parser*.
- `sequence` as an object consisting of one or more blocks, followed obligatorily by `SEQUENCE_END`. It represents the "grammar transposition" of the single test sequence.
- `block` as an object consisting of a `BLOCK_START`, an element of a specific collection of "syntactic elements", and a `BLOCK_END`.

```
'''
      //################# sentence structure#################

      start: sequence+
      sequence: block+ SEQUENCE_END
      block: BLOCK_START (\ [....]) BLOCK_END

'''
```

Having declared the macro structure, it is possible to cascade the **syntax**: the conversion into specific **tokens** of certain portions of the string analysed in *input*.

```
'''
     // ################ Blocks syntax ###################

     block_led_on: "led_on" SEPARATOR

     block_led_off: "led_off" SEPARATOR

     block_sum: "sum" SEPARATOR var1_digit SEPARATOR var2_digit
```

```
SEPARATOR

     block_subtraction: "subtraction" SEPARATOR var1_digit
SEPARATOR var2_digit SEPARATOR

     block_multiplication: "multiplication" SEPARATOR
```

The *parser* will then proceed to generate a **Syntactic Tree** which is equivalent to a syntactic representation of the structure of the text document, in accordance with the previous grammar rules.

According to the *syntax* [10] decided for the Discord message:

```
# [block-name]; var1:[value]; var2:[value]; end
```

a **token** was implemented for each 'function-block' in the following form:

```
block_[block_name]: "[block_name]" SEPARATOR var1_digit
SEPARATOR var2_digit SEPARATOR
```

To insert new 'function-blocks' only need to enter the **associated token** in the indicated form.

ParserLogic.py

The **logic** defines how the *input* language is to be manipulated! In this file, the superclass `ParserLogic` has been implemented: it inherits from the `Transformer` class (of the lark library) various methods such as `transform()`.

By invoking this function, the **Syntactic Tree** will be processed from the leaves to the root, and at each node the function with the same name defined in the `ParserLogic` class will be executed. The results produced by the i-th node are then processed from time to time and returned to the higher node: this process ends once the root is reached where the final result is stored within the class itself.

For each *log file*, the *parser* will store as a final object a list of test sequences, already coded as expected at project level:

```
[sequence number][function-block
encoding][function-block-mode encoding][value var1][value
var2][value var3][function-block position within the seq.]
```

To insert new 'function-blocks' ONLY need to define the new block logic in this class.

ParserTools.py

In this file, the `ParserTools` class has been implemented: it provides the user with the main functionality needed to interact with this **module**. Using the methods `process_files` and `process_folder`, the searcher can select the log files to be processed and execute the entire parsing algorithm. They will automatically start the process of reading and encoding the text files given as input to the system, and will output **two files per logfile**:

- `.sv` file containing the logical translation of the syntactic tree corresponding to the logfile

- `.png` file containing the visual translation of the syntactic tree corresponding to the logfile

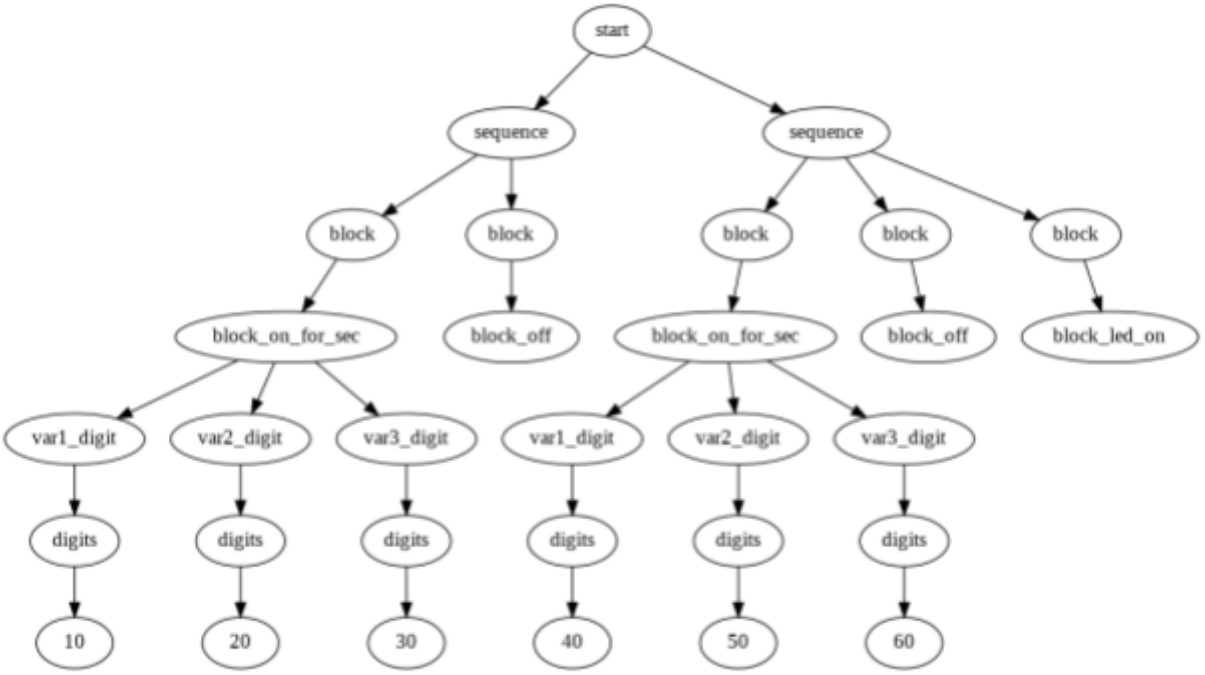


Figure 15: Parser-tree obtained after processing the following logfile:

```
# OnForSec;var1:10;var2:20;var3:30;end#Off;end##
# OnForSec;var1:40;var2:50;var3:60;end#Off;end#led_on;end##
```

2.4.3. `features_matrix_module`

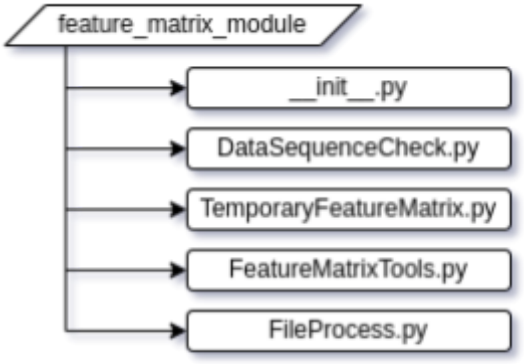


Figure 16: feature_matrix_module structure

Once the log files have been encoded by the parsing process, this **module** handles the manipulation and extraction of the information required to generate the **feature matrix** (according to the **Supervised approach**).

`features_matrix_module` requires `ParserGrammar.py` and `ParserLogic.py` from `parser_module`; it also imports `panda` and `numpy` Python libraries. The entire code was segmented into the following Python files:

- `DataSequenceCheck.py`
- `TemporaryFeatureMatrix.py`
- `FileProcess.py`
- `FeatureMatrixTools.py`

each of them implements a specific module **component** and allows targeted modifications without compromising the functionality of the entire structure.

DataSequenceCheck.py

This file provides the `DataSequenceCheck` class: it is a container of useful sequence information within a single log file. Referring to the first experiment and [11], the collection of info related to exercise A has already been implemented:

- `motors`, number of motor-blocks occurring in a test sequence.
- `loops`, number of loop-blocks occurring in a test sequence.
- `conditionals`, number of conditional-blocks occurring in a test sequence.
- `others`, number of other-blocks occurring in a test sequence.

The collection of new information, more useful in future contexts, can be extended by modifying this simple class.

```
class DataSequenceCheck():
 def __init__(self,motors,loops,conditionals,others):
   # Exercise A categories
   self.motors = motors
   self.loops = loops
   self.conditionals = conditionals
   self.others = others
```

Instances of the class `DataSequenceCheck` are managed by the class `FileProcess`.

TemporaryFeatureMatrix.py

This file provides the `TemporaryFeatureMatrix` class: it represents the encoding of the information contained in a log file through specific **indicators**. According to [11], the values associated with each indicator are the result of the comparison between two contiguous test sequences, and they characterise the **Supervised approach.** In the ToolBox only the main indicators, common to a generic type of exercise, have been structured:

- **Same:** represents the number of equal blocks, compared to the previous sequence; to calculate this indicator, a function counts how many blocks with the same `[block-function encoding]`, same `[block-function-mode encoding]` and exactly the same parameters are present comparing two contiguous sequences.

- **Added:** represents the number of blocks added compared to the previous sequence; to calculate this indicator, a function checks two contiguous sequences and updates the added parameter if there are differences in `[block-function encoding]` or `[block-function-mode encoding]`.

- **Modified:** represents the number of blocks that have been changed from the previous sequence; to calculate this indicator, a function counts how many blocks with the same `[block-function encoding]` and the same `[block-function-mode encoding]` but different parameters are present comparing two contiguous sequences.

- **Deleted:** represents the number of blocks deleted from the previous sequence; to calculate this indicator, a function counts how many blocks in a specific sequence were deleted in the consecutive sequence (only if they are not classified as **Modified**).

- **Delta Motors:** the amount of change in the motor block parameters (first, second or third one), compared to the previous sequence (calculated only for **Modified** blocks); the ***Euclidean norm*** is used to calculate this indicator and all these values are added together to obtain delta motors.

- **Delta Loops:** the amount of change in the parameters of the **Loop blocks**, compared to the previous sequence; the procedure is the same as that explained for **Delta Motors** but considering the **Loop** category.
- **Delta Conditionals:** the amount of change in the parameters of the **Conditional blocks**, compared to the previous sequence; the procedure is the same as that explained for **Delta Motors**, but considering the **Conditional** category.
- **Delta Others:** the amount of change in the parameters of the **Other blocks**, compared to the previous sequence; the procedure is the same as that explained for **Delta Motors**, but considering the **Other** categories.

As with the `DataSequenceCheck`, the `TeamporaryFeatureMatrix` class also acts as a container and is used by the `FileProcess` class.

The matrix contained here, which can be extracted using the `.get()` method, does not yet represent the desired **feature matrix**: mean and std. deviations of the various indices have not yet been calculated.

FileProcess.py

One of the main classes of this module is defined in this file: `FileProcess` handles the procedures needed to evaluate the data needed to set the **TemporaryMatrix** and the **DataSequenceCheck.** Describing the main methods:

- `process_data_sequence_checks()`: analyses each test sequence and extrapolates its "checks"
- `process_temporary_matrix()`: makes comparisons between two contiguous sequences, extrapolating their **indicators** values, calling the

homonymous calculation methods. Having defined the **i-th sequence** and the sequence of **index i-1** as `current` and `previous` respectively, the following functions are used:

- `same()`:
- `modified()`:
- `added()`.
- `deleted()`.
- `delta_mototrs()`.
- `delta_loops()`.
- `delta_conditionals()`.
- `delta_conditionals()`.
- `delta_others()`.

The computation of the various deltas is performed using the **Euclidean distance** and selecting only the **minimum delta** of each test sequence.

- `process()`: performs both of the above functions.

There are cases where a specific delta cannot be calculated (absence of a block type) and the algorithm assigns a **NaN** value. It creates no problems in the evaluation of **mean** and **std. Deviation**.

Below is a portion of the code used to evaluate **Delta Motors**:

```
 def delta_motors(previous,current):
     delta_motors = None
     for i in range(0,len(current)):
       current_block = current[i]
       min_delta = None
       for y in range(0,len(previous)):
         previous_block = previous[y]
         # Checking if the current sequence block has the same
name and mode of the previous sequence block
         if current_block[1] == previous_block[1] ==
int(BlocksEnum.MOVE_STEERING) and current_block[2] ==
previous_block[2]:
           # Evaluate the delta between two blocks of two
```

```
contiguous sequences
            current_block_np = np.array(current_block)
            previous_block_np = np.array(previous_block)
            current_delta = np.linalg.norm(current_block_np[3:-1] -
previous_block_np[3:-1])
            if min_delta != None:
              # If min_delta has a value (it's not equal to None),
              # assign min deltha the minimum value between the
previous and current min deltha
              min_delta = min(min_delta,current_delta)
            else:
              # Otherwise assign min_delta the current value
              min_delta = current_delta
        if min_delta != None:
          if delta_motors == None:
            delta_motors = 0
          delta_motors += min_delta

      if delta_motors == None:
        delta_motors = np.NaN
    # If no comparisons have been made,
    # delta_motors remains equal to None,
    # and it will be shown as NaN in the table
    return delta_motors
```

If new indicators are introduced for the two data structures described above, the corresponding calculation algorithm must be included within this class.

FeatureMatrixTools.py

In this file, the `FeatureMatrixTools` class has been implemented: it provides the user with the main functionality needed to interact with this **module**. This class has been implemented considering that the **parser** and the `FileProcess` class will operate for several log files.

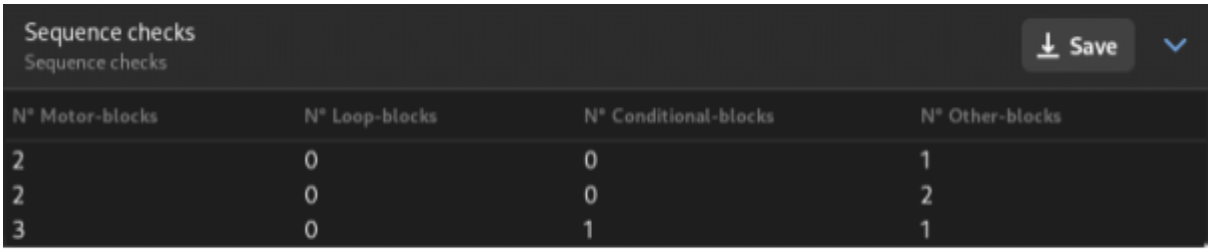
The `.process_files()` method starts the procedure for calculating the various indicators described above and the feature matrix. This function requests as an argument a list of tuples containing the **path** of the log file and the **label** associated with it.

```
 def process_files(self, path_lable_list):

   if not isinstance(path_lable_list, list):
     path_lable_list = [path_lable_list]

   for tupla in path_lable_list:
     if len(tupla) !=2:
       raise TypeError("The dimension of the tuple {} is not equal to
2".format(tupla))

     if not isinstance(tupla[0], Path):
       raise TypeError("The first tuple member is not a Path type")

     if not isinstance(tupla[1], int):
       raise TypeError("The second tuple member is not int type")


     if not tupla[0].is_file():
       print("Il path {} non esiste o non è un file. File ignorato,
continuo.".format(tupla[0]))
       continue

     self.labels_vector["Labels"].append(tupla[1])
     #print(self.labels_vector["Labels"])
     data = tupla[0].read_text()
     parse_tree = parser_grammar.parse(data)

     #self.graph_tree(parse_tree, path)
     parser_logic = ParserLogic()
     parser_logic.transform(parse_tree)
     file_process = FileProcess(parser_logic.analyzed_file)
     file_process.process()


self.data_sequence_checks.append(file_process.get_data_sequence_checks()
)
     self.temporary_matrixes.append(file_process.get_temporary_matrix())

   self.create_final_feature_matrix()
```

The `data_sequence_checks`, `temporary_matrices` and `final_feature_matrix` can be displayed using the functions `.show_sequences_info_and_temporary_matrices()` and `.show_feature_matrix()`

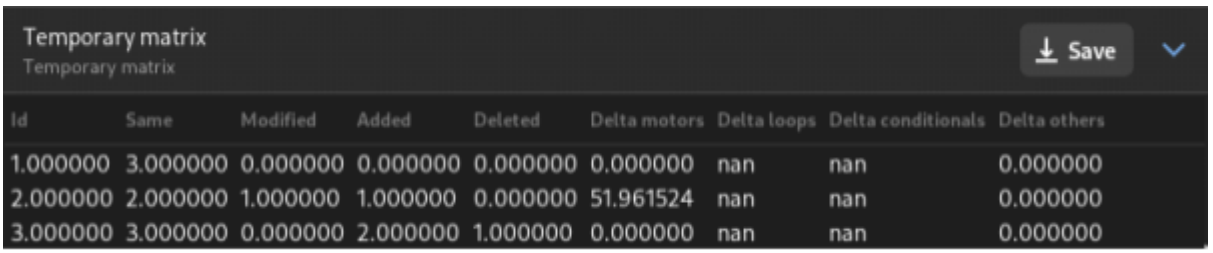
Below are the results obtained from processing the following log file:

```
#OnForSec;var1:10;var2:20;var3:30;end#Off;end#led_on;end##
#OnForSec;var1:40;var2:50;var3:60;end#Off;end#led_on;end#led_off;end##
#OnForRotation;var1:10;var2:20;var3:30;end#Off;end#IFDO;var1:CondizioneD
aValutare;end#OnForSec;var1:40;var2:50;var3:60;end#led_off;end##
```



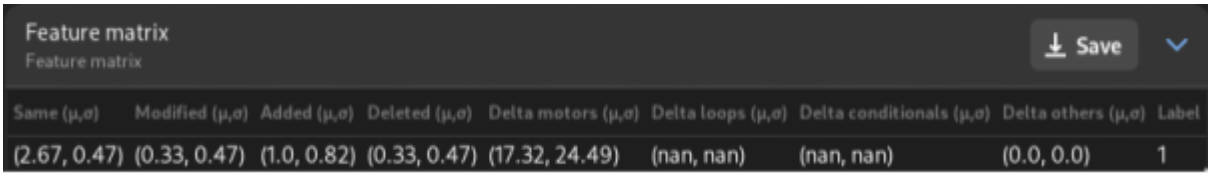
N° Motor-blocks	N° Loop-blocks	N° Conditional-blocks	N° Other-blocks
2	0	0	1
2	0	0	2
3	0	1	1

Figure 17: GUI display of Sequence checks



Id	Same	Modified	Added	Deleted	Delta motors	Delta loops	Delta conditionals	Delta others
1.000000	3.000000	0.000000	0.000000	0.000000	0.000000	nan	nan	0.000000
2.000000	2.000000	1.000000	1.000000	0.000000	51.961524	nan	nan	0.000000
3.000000	3.000000	0.000000	2.000000	1.000000	0.000000	nan	nan	0.000000

Figure 18: GUI display of Temporary matrix



Same (μ,σ)	Modified (μ,σ)	Added (μ,σ)	Deleted (μ,σ)	Delta motors (μ,σ)	Delta loops (μ,σ)	Delta conditionals (μ,σ)	Delta others (μ,σ)	Label
(2.67, 0.47)	(0.33, 0.47)	(1.0, 0.82)	(0.33, 0.47)	(17.32, 24.49)	(nan, nan)	(nan, nan)	(0.0, 0.0)	1

Figure 19: GUI display of Feature Matrix

The feature matrix also contains the labels associated with each log file; it can be saved as a `.csv` file via the `.save_feature_matrix()` command to be used as input for machine learning algorithms in the future. Below the `.csv` file associated to previous log file:

```
,Same,Modified,Added,Deleted,Delta Motors,Delta Loops,Delta
Conditionals,Delta Others,Labels
File 1,"[2.6666666666666665,
0.4714045207910317]","[0.3333333333333333,
0.4714045207910317]","[1.0,
0.816496580927726]","[0.3333333333333333,
0.4714045207910317]","[17.320508075688775,
24.494897427831777]","[nan, nan]","[nan, nan]","[0.0, 0.0]",1
```

2.4.4. `machine_learning_module`

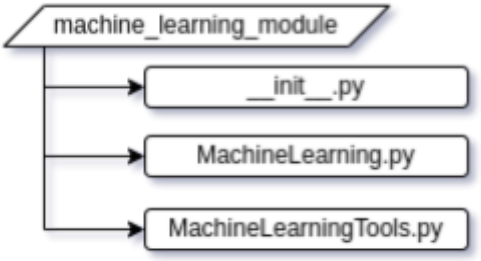


Figure 20: machine_learning_module structure

This module helps the researcher to manage the instantiation, training and testing phases of the four machine learning models selected by the project authors [3]:

- **Support Vector Machine (SVM)**: a widely used supervised linear classification method, it can be considered as an extension of **Perceptron**.
- **Logistic Regression (LR):** a linear model for binary classification that can be extended to multiclass classification using the **One vs Rest (OvR) technique.** It originated in the field of **Statistical Mathematics** and is based on the use of the **Logistic Function** as its activation function.
- **K-Nearest Neighbour classifier (K-NN)**: a special supervised learning method that differs substantially from classical classification models. Using input data, it attempts to store the entire dataset instead of deriving and refining the parameters of a decision function.
- **Random Forest (RF):** one of the most popular machine learning methods due to its good computational performance, scalability and easy use. From a logical-structural point of view, this learning algorithm can be seen as a set of **decision trees**: by combining weaker classification systems, it is possible to obtain a robust model that is less prone to overfitting.

The proposed **module** requires the following Python libraries:

- `sklearn`: to manage machine learning models.
- `numpy` and `pandas`: to manipulate the training matrices and results produced by the models.
- `pathlib`: for loading/saving fitted machine learning models.
- `matplotlib.pyplot`: to graph significant results related to the testing phase of the model.

and the entire code is segmented into two Python files: `MachineLearning.py` and `MachineLearningTools.py.`

MachineLearning.py

This file provides the `MachineLearning` class, which implements all useful functions for manipulating **models** and their **inputs** and **outputs**. During the initialisation, this class provides the user with the four machine learning models, set with standard parameters. Two additional versions named `_pca_2D_model` and `_pca_3D_model` are associated with each model, and they will be trained with the reduced feature matrix via the **Principal Component Analysis (PCA)** method.

```
 # rf models
 rf_model = RandomForestClassifier(criterion='entropy',n_estimators=10,
random_state=1, n_jobs=2)
 rf_pca_2D_model =
RandomForestClassifier(criterion='entropy',n_estimators=10,
random_state=1, n_jobs=2)
 rf_pca_3D_model =
RandomForestClassifier(criterion='entropy',n_estimators=10,
random_state=1, n_jobs=2)
```

```
 # knn models
 knn_model = KNeighborsClassifier(n_neighbors=5, p=2,
metric='minkowski')
 knn_pca_2D_model = KNeighborsClassifier(n_neighbors=5, p=2,
metric='minkowski')
 knn_pca_3D_model = KNeighborsClassifier(n_neighbors=5, p=2,
metric='minkowski')

 # svm models
 svm_model = SVC(kernel='linear', C=1.0, random_state=0)
 svm_pca_2D_model = SVC(kernel='linear', C=1.0, random_state=0)
 svm_pca_3D_model = SVC(kernel='linear', C=1.0, random_state=0)

 # lr models
 lr_model = LogisticRegression(C=1000.0, random_state=0, max_iter=100)
 lr_pca_2D_model = LogisticRegression(C=1000.0, random_state=0,
max_iter=100)
 lr_pca_3D_model = LogisticRegression(C=1000.0, random_state=0,
max_iter=100)

 # PCA models settings
 pca_2D_model = PCA(n_components=2)
 pca_3D_model = PCA(n_components=3)
```

In addition to the machine learning models, the `__init__()` function sets to `None` a list of parameters useful for collecting data from the **preprocessing**, **training** and **testing phases** of the algorithms.

```
def __init__(self):
   self.loaded_feature_matrix = None
   self.loaded_labels_vector = None

   self.std_training_feature_matrix = None
   self.std_test_feature_matrix = None

   self.training_labels_vector = None
   self.test_labels_vector = None

   self.training_feature_matrix_pca_2D_model = None
   self.training_feature_matrix_pca_3D_model = None

   self.test_feature_matrix_pca_2D_model = None
   self.test_feature_matrix_pca_3D_model = None
```

```
self.loaded_model = {
    "real_model":None,
    "pca_2D_model":None,
    "pca_3D_model":None
}
self.trained_model = {
    "real_model":None,
    "pca_2D_model":None,
    "pca_3D_model":None
}

self.model_prediction = None
self.pca_2D_model_prediction = None
self.pca_3D_model_prediction = None
```

Once a MachineLearning object has been instantiated, the user can load the **feature matrix** and its associated **labels vector** using the `.load_data()` method. This data can be passed explicitly or extracted from a `.csv` file: this data may be passed explicitly or extracted from a .csv file: in the second case, the system will convert the file into a `DataFrame` and associate the **labels vector** with the column named "*Labels*", and the **feature matrix** with the rest of the data. Before loading the data passed by the user, the `.load_data()` method will perform some important dimensionality checks to comply with the input requirements of the **sklearn classifiers**.

The loaded data are **preprocessed** by the system before being used in the training and/or testing phases. Using the function `.preprocessing_loaded_feature_matrix()`, the user specifies to the system how the feature matrix should be used: as input for a training, or as input for a test, or both. Then the system proceeds to manipulate the feature matrix according to the chosen mode. At this stage, **PCA reduction** is performed to collect two- and three-dimensional versions of the same feature matrix. These will later be used to give a qualitative evaluation of the model's accuracy during the test phase. The **PCA**

methods are provided by the `sklearn` library and refer to the class implemented by Minka [19] and the probabilistic models of Tipping and Bishop [20].

Even models can be loaded to perform further training or simply test their accuracy: the system will check if the type of the loaded model corresponds to one of those described above and, in case it does, it will also attempt to automatically load its `_pca_2D` and `_pca_3D` versions. Below is a portion of code describing the loading of a `RandomForestClassifier`:

```
if isinstance(loaded_model,RandomForestClassifier):

     model_path_2D =
Path(str(model_path).split(".joblib")[0]+"_pca_2D.joblib")
     model_path_3D =
Path(str(model_path).split(".joblib")[0]+"_pca_3D.joblib")

     if model_path_2D.is_file():
       loaded_pca_2D_model = load(model_path_2D)
     else:
       print("2D pca model version not found")
       loaded_pca_2D_model = None

     if model_path_3D.is_file():
       loaded_pca_3D_model = load(model_path_3D)
     else:
       print("3D pca model version not found")
       loaded_pca_3D_model = None

     self.loaded_model["real_model"] = loaded_model
     self.loaded_model["pca_2D_model"] = loaded_pca_2D_model
     self.loaded_model["pca_3D_model"] = loaded_pca_3D_model
```

In the **training phase**, the user can use the `.training_model()` method for a previously loaded model or a new one. In the second case, it is necessary to specify which type of model should be initialised from the four proposed: `["svm","lr","rf","knn"]`. The model is trained by providing as input the

`std_training_feature_matrix` and the `training_labels_vector`; at the same time, also its reduced versions are trained via PCA. The results produced by the training phase are saved within the class in the **dictionary** `trained_model`.

The user can save the newly trained model locally using the `.save_model()` function: it will encode the sklearn classifier in a `.joblib` file with a personalised name. The versions `_pca_2D` and `_pca_3D` are also automatically saved in the same way by adding the extension `_pca_2D` and `_pca_3D` respectively to the name chosen by the user.

In the **testing phase**, the `.model_predict()` function uses the `std_test_feature_matrix` to obtain the model prediction as a vector of labels and collect it. The user can later compare them with the `test_labels_vector` and find out how many misclassifications the model performed. Many useful info about the model performances (i.e. `misclassification_model` and `error_percentage_model`) can be obtained using the method `.misclassification()`.

```
 def misclassification(self):

   misclassification_model = None
   error_percentage_model = None
   misclassification_pca_2D_model = None
   error_percentage_pca_2D_model = None
   misclassification_pca_3D_model = None
   error_percentage_pca_3D_model = None

   if type(self.model_prediction) == type(None) or
type(self.test_labels_vector) == type(None):
     raise Exception("Missing model_prediction or
test_labels_vector")

   misclassification_model = sum(1 for i, j in
zip(self.model_prediction, self.test_labels_vector) if i != j)
   error_percentage_model = misclassification_model /
len(self.test_labels_vector) * 100
```

```
   if type(self.pca_2D_model_prediction) != type(None):
     misclassification_pca_2D_model = sum(1 for i, j in
zip(self.pca_2D_model_prediction, self.test_labels_vector) if i !=
j)
     error_percentage_pca_2D_model =
misclassification_pca_2D_model / len(self.test_labels_vector) *
100

   if type(self.pca_3D_model_prediction) != type(None):
     misclassification_pca_3D_model = sum(1 for i, j in
zip(self.pca_3D_model_prediction, self.test_labels_vector) if i !=
j)
     error_percentage_pca_3D_model =
misclassification_pca_3D_model / len(self.test_labels_vector) *
100

   return (misclassification_model,
           error_percentage_model,
           misclassification_pca_2D_model,
           error_percentage_pca_2D_model,
           misclassification_pca_3D_model,
           error_percentage_pca_3D_model)
```

MachineLearningTools.py

This file provides the `MachineLearningTools` class, which implements all useful functions to plot qualitative graphs regarding the model's classification ability. Some of the methods defined in this class are:

- `plot_explained_variance()`: shows the amount of information associated with the n-eigenvectors obtained from the PCA decomposition (*Figure 21*)

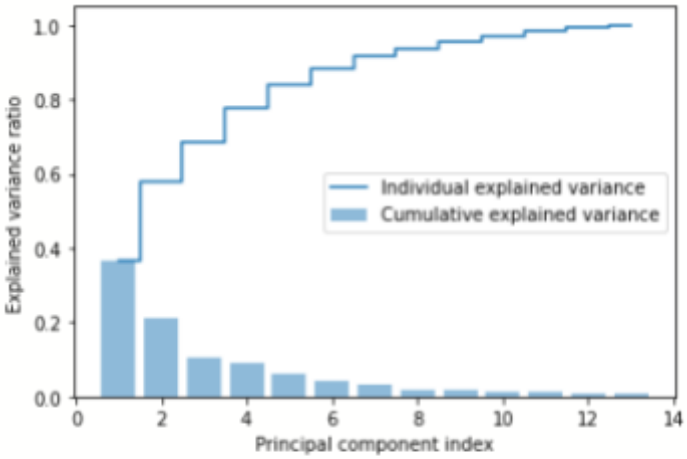


Figure 21: Example of `plot_explained_variance()`

- `plot_3D_classification()`: graphics of the feature matrix samples in a reduced 3-dimensional space. Each element is coloured according to the label predicted by the model. (*Figure 22*)

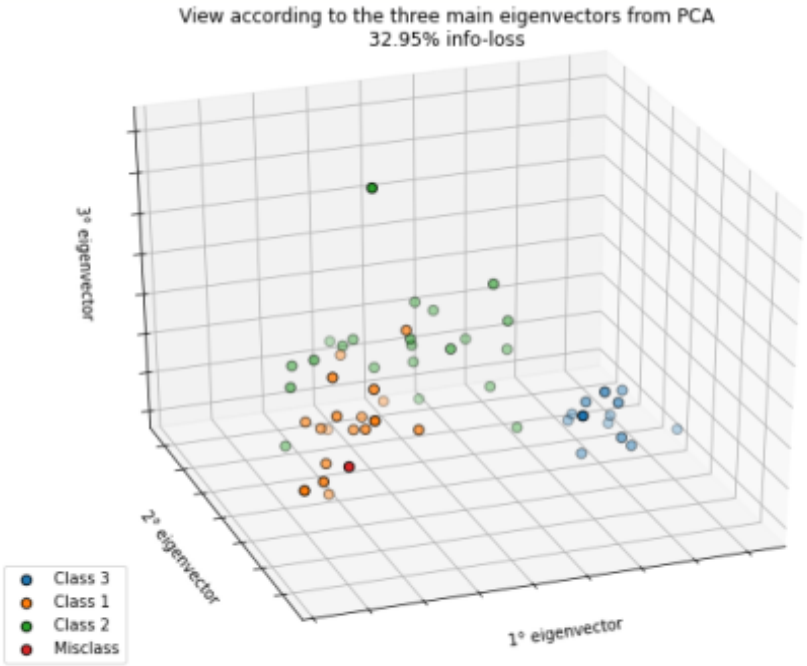


Figure 22: Example of `plot_3D_classification()`

- `plot_2D_decision_region()`: graphics of the feature matrix samples in a reduced bidimensional space. Each element is coloured according to the label

predicted by the model, and the space is divided according to the decision regions. (*Figure 23*)

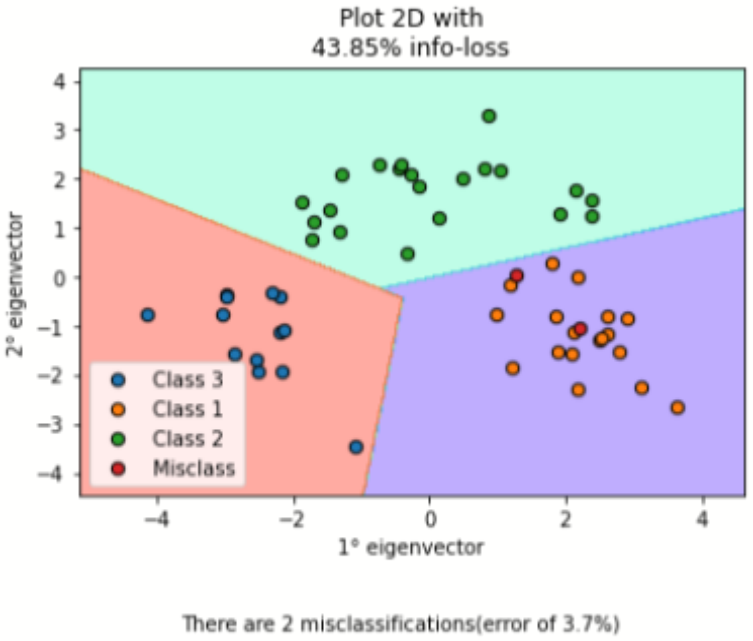


Figure 23: Example of `plot_2D_decision_region()`

2.4.5. Graphic interface

The MLToolBox GUI is designed to allow graphical interaction with the modules described above. For its development, the **Gtk toolkit** was used through the template method: this facilitates the integration between the gtk definition file (graphic structure of the page) and the Python code (page interactions).

Each page is organised into at least two files:

- `.py`: the interactions with the page are defined. They are structured with a python class decorated with a `Gtk.Tamplate` *(Figure 24)*
- `.ui`: the hierarchy of the widgets to be displayed is defined. *(Figure 25)*

Pages that have dynamically generated content (e.g. results and/or graphics) can utilise sub-pages to improve code management and modularity.

Page general structure

Within the main window, each module is represented by a dedicated page structured graphically as follows:

- **Module logo**
- **Module name**
- **Introductory description** of the module
- **Module interaction section**: in this section it is possible to set the parameters required to use the module's functionalities, and to interact with the module itself. Each parameter is provided with a short description.

The module results are displayed dynamically at the bottom of the page and can be saved locally. In the case of a new execution, results already shown will be overwritten. It is possible to detach the results of an execution in a secondary window in order to store and compare them with the results of future executions.

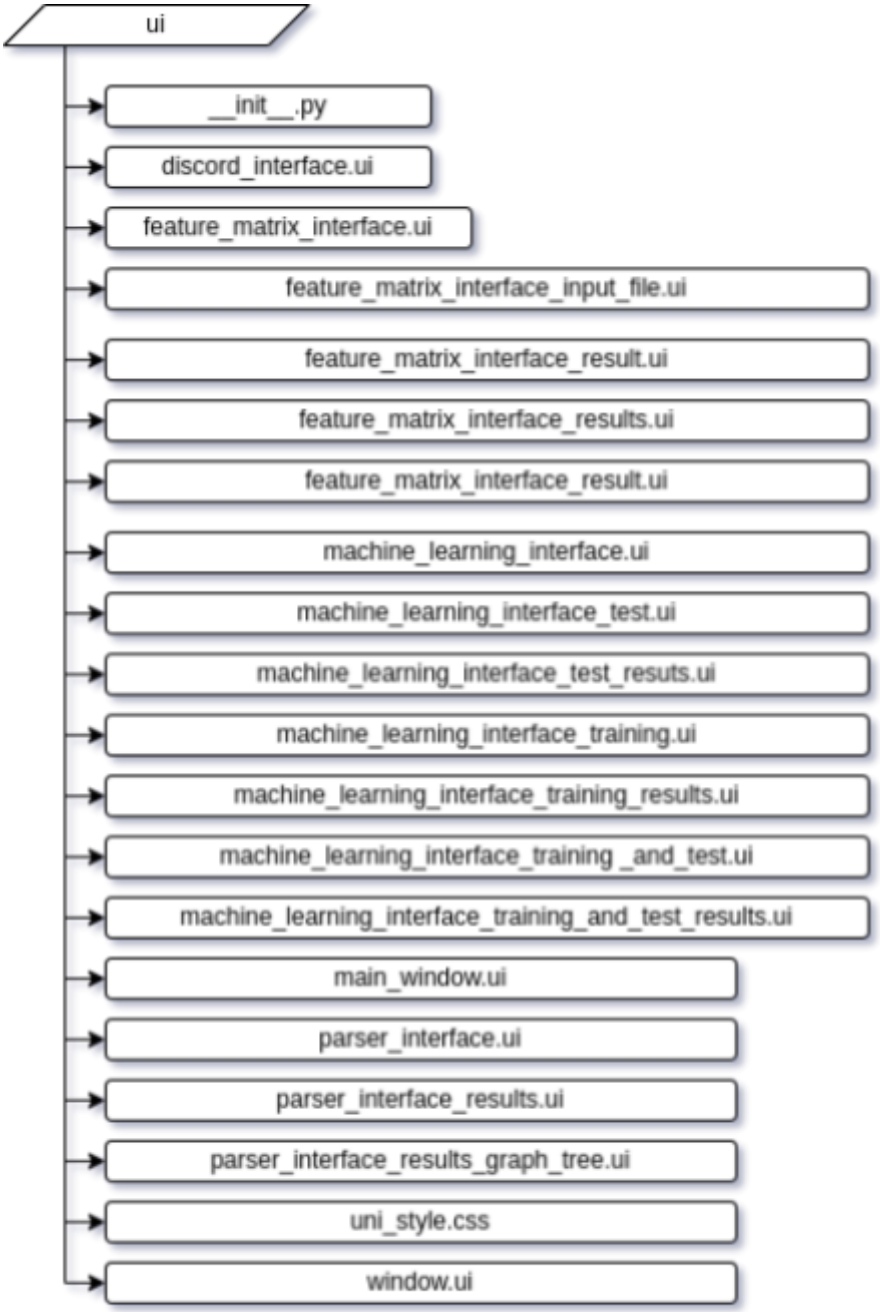


Figure 24: list of .ui files

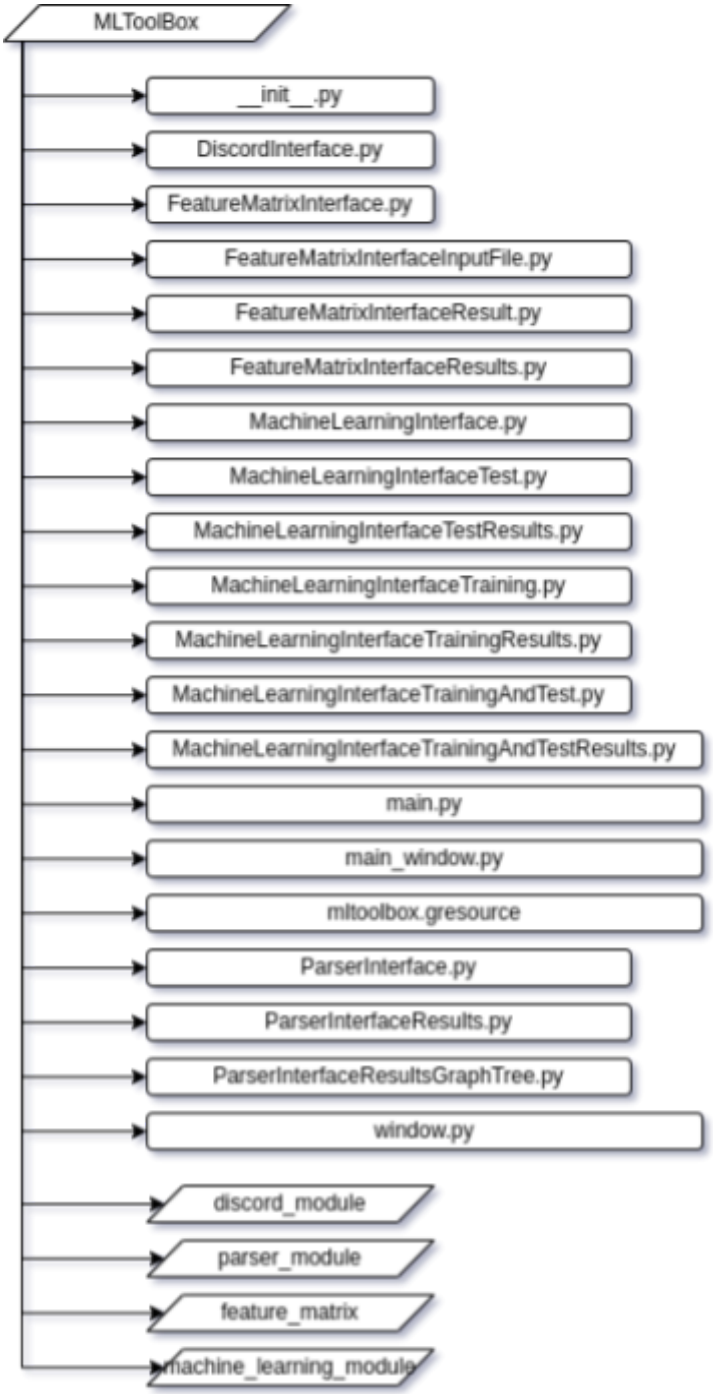


Figure 25: list of .py files

Discord page

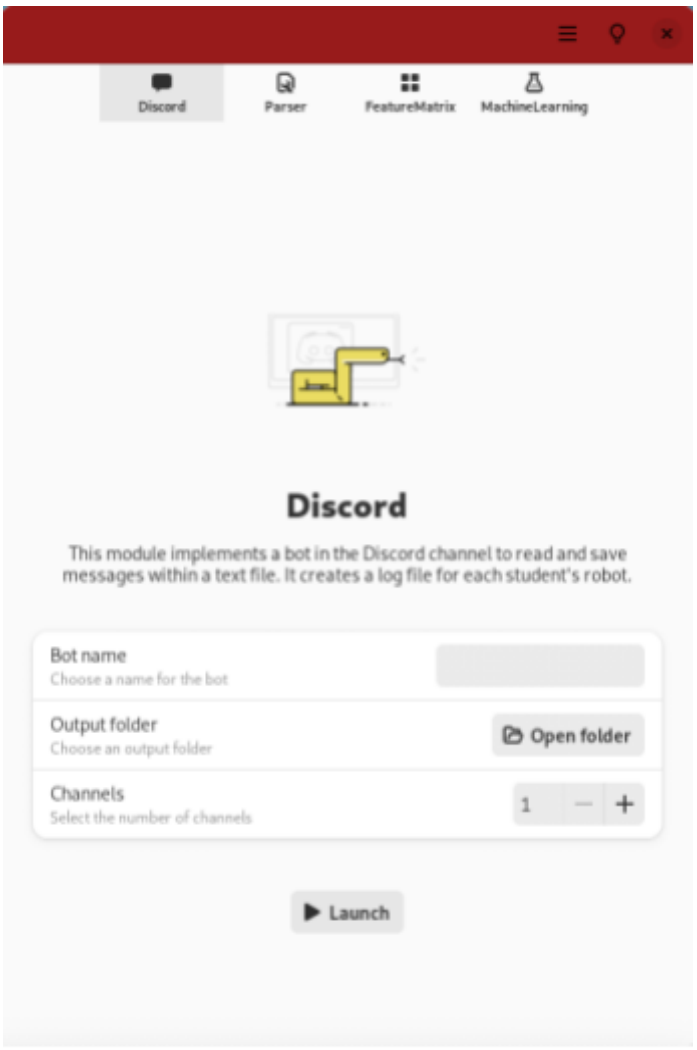


Figure 26: Discord page

On this page it is possible to set the bot's initialisation parameters and manage its execution. The user can associate a name to the bot, select the folder in which the bot collects log files and set the number of channels to be created. The bot's execution button highlights its state transitions: as long as the bot is running, parameter selection is disabled.

Parser page

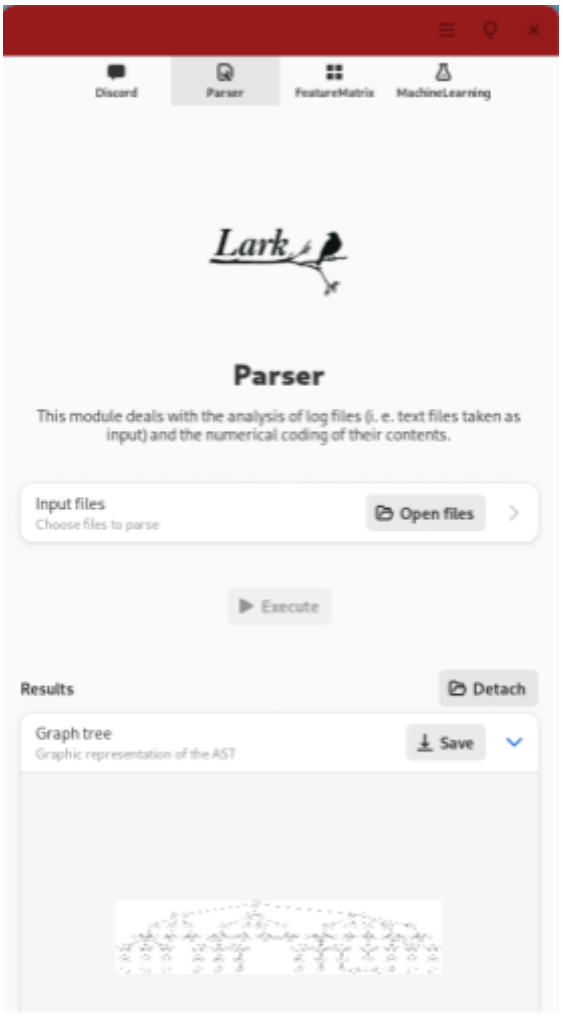


Figure 27: Parser page

On this page, it is possible to manage log files parsing. The user can select one or more text files and process them using the **Execute** button. The selected files are displayed dynamically on the interface and can be removed from the list. At the end of execution, the graph-tree associated with each file will be displayed and can be saved locally.

FeatureMatrix page

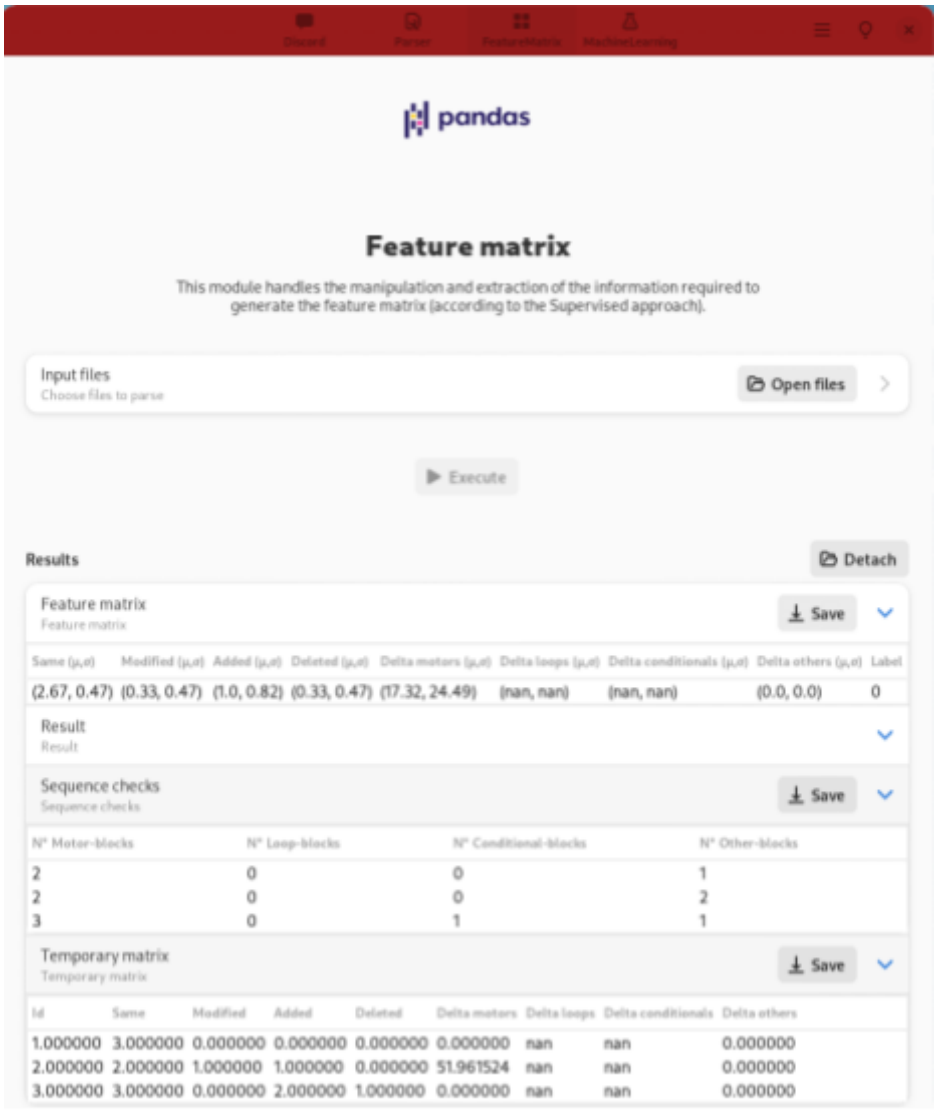


Figure 28: FeatureMatrix page

On this page it is possible to manage the feature matrix creation phase. The user selects the log files to be processed, associating the corresponding label/classification with each of them (Supervised approach [3]). Like the parser page, selected files are dynamically displayed on the interface and can be removed from the list.

Once the execution of the module is complete, the following results will be shown:

- The feature matrix
- Datasequencecheck and Temporarymatrix associated with each file.

All results can be saved as a `.csv` file.

MachineLearning page

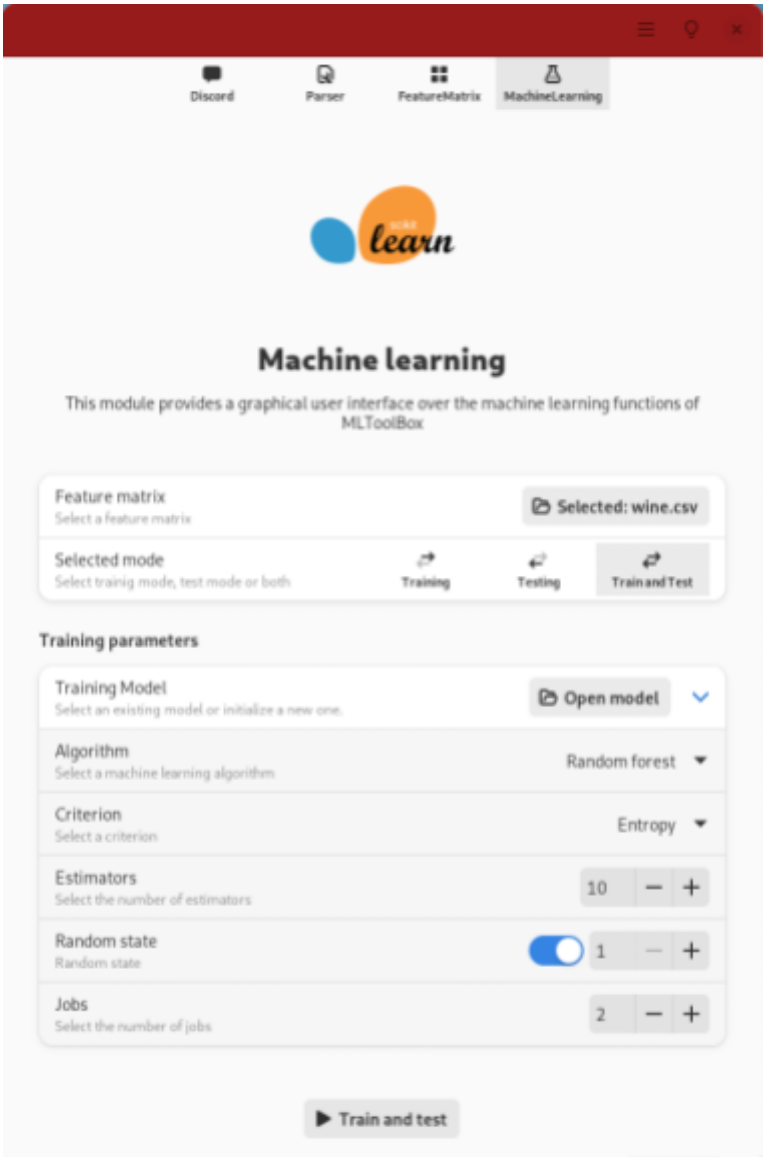


Figure 29: MachineLearning page

On this page, it is possible to manage the `machine_learning_module`. The GUI provides the ability to load the feature matrix from a csv file and to choose the data processing purpose. Selecting a specific mode from the proposed three (i. e. "Training," "Testing" and "Train and Test") will dynamically display the associated functionalities. The user can decide to load a model or initialise a new one, specifying its type and configuration parameters. Once all values required by the interface have been configured, starting the execution of the system will generate and visualise the results of the processing.

The results displayed are of two categories:

- **Training results**, constituted by the model trained with the feature matrix given as input. It can be locally saved by the user as a `.joblib` file.

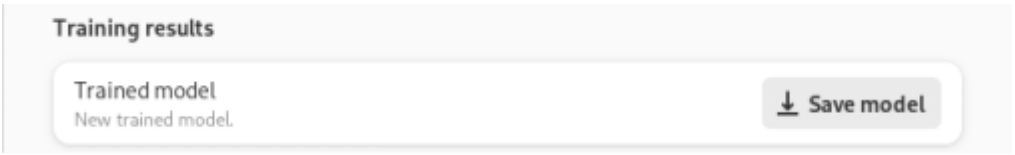


Figure 30: MachineLearning page, Training results

- **Prediction results**, constituted by the model's output values and qualitative graphs regarding its classification accuracy. All of them can be saved locally by the user respectively as `.csv` files and `.png` images.

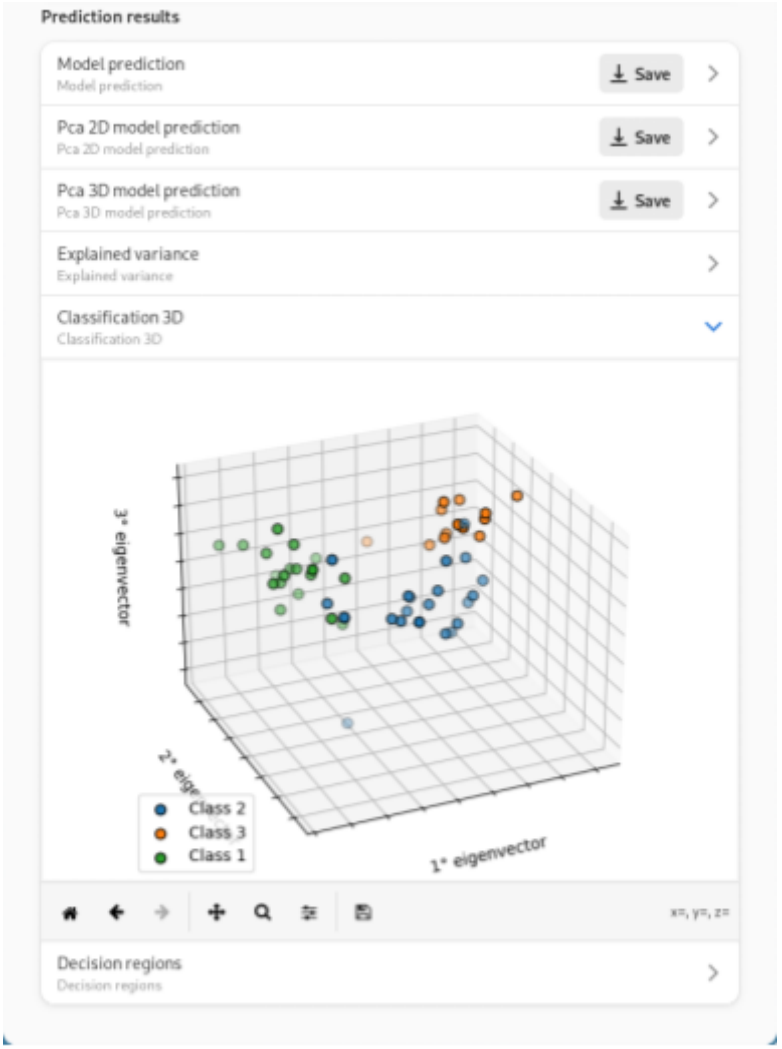


Figure 31: MachineLearning page, Prediction results

2.4.6. Packaging

The internal file structure of the project is designed to generate a **Python Wheel** and a **Flatpak package**: in addition to the program scripts, there are also the necessary files for both packaging processes. Future modules, scripts and files must be included in a way that they will be recognised as elements of the package.

Python Wheel specific files:

- `setup.py`: This file dictates how the Python package needs to be built. It is configured to also generate a new terminal command, `mltoolbox`, that will launch the user interface. Before starting the packaging process, the `mltoolbox.gresources.xml` is compiled into its binary form as `mltoolbox.gresources` and is automatically updated and collected into the package.

Flatpak specific files:

- `meson.build`: **meson** is the build-toolchain used to generate this Flatpak package. Multiple folders contain a file named `meson.build` used during the Flatpak package compilation.
- `org.example.App.json`: This file contains all the fundamental information required to build the Flatpack package (i.e. application name, build requirements and permissions). It will also include the dependencies bringed by the `python3-modules.json` and `graphviz.json` files.
- `python3-modules.json`: This file contains additional Python dependencies required to be installed into the Flatpak package.
- `graphviz.json`: This file describes where to retrieve and how to compile `graphviz`, a library required to produce visual representation for graph structures.

Common files:

- `mltoolbox.gresources.xml`: This file describes a list of resources (e.g. images and definitions of the GtkBuilder user interface) to be collected in a single binary file. This binary file is then loaded into the code, simplifying resource management.
- `src/icons`: This folder contains all the icons used into the project.
- `src/ui`: This folder contains all the GtkBuilder UI definitions required to build the graphical user interface.
- `src/MLToolBox`: This folder contains all the Python scripts required to run the interface.
- `src/MLToolBox/discord_module`: This folder contains all the Python scripts related to the discord module.
- `src/MLToolBox/parser_module`: This folder contains all the Python scripts related to the parser module.
- `src/MLToolBox/feature_matrix_module`: This folder contains all the Python scripts related to the feature matrix module.
- `src/MLToolBox/machine_learning_module`: This folder contains all the Python scripts related to the machine learning module.
- `src/MLToolBox/mltoolbox.gresources`: This is the result of compiling the `mltoolbox.gresources.xml` in a binary format.

3. Conclusions

The goal achieved with the realisation of the MLToolBox was to provide automated and helpful tools to carry out ER-related research experiments. The system is able to summarise and manage the large amount of data processing related to each phase in just a few commands, facilitating and supporting the researchers in their analysis. This system is a prototype and can be improved by introducing numerous other features, but the presence of a well-structured graphical support interface makes it already usable within the dynamics of a real experiment. Certainly an experienced programmer can gain advantages from using this tool by programming directly from the terminal, but the functionality offered by the GUI is still sufficient to guarantee a valid support. The MLToolBox can be seen as a first step towards helping teachers to collect data independently during an ER course, helping researchers and project authors to work on a larger scale.

3.1. Future works

The structure given to the MLToolBox is designed to ensure that its modules are easy to read, understand and extend. Future developers will be able to add functionality or improve existing ones, as long as they handle the dependencies with other modules and the general structure of the system. Each new feature given to the toolbox must then be matched graphically by updating the GUI.

At the moment, the prototype is only able to perform an analysis according to the **Supervised approach** [3]. A first future improvement may be to extend the `feature_matrix_module` by adding data processing according to the **Mixed approach** guidelines [3].

Other suggested improvements are:

- Extend the grammar accepted by the parser allowing the student to code the M5 Stick with new block-functions.
- Add class-methods to compare the performance offered by the various trained machine learning algorithms.
- Create a module dedicated to the use of neural networks in order to characterise the students' behaviour by **deep learning** tools.

The MLToolBox will make it possible to improve data collection and thus reach new conclusions regarding the relationship between ER activities and student learning.

References

[1] *D. Scaradozzi, L. Screpanti, and L. Cesaretti, "Towards a definition of educational robotics: a classification of tools, experiences and assessments," in Smart Learning with Educational Robotics, L. Daniela, Springer, Cham, 2019.*

[2] *F. B. V. Benitti, "Exploring the educational potential of robotics in schools: A systematic review, "Computers & Education, vol. 58,no. 3, pp. 978-988, 2012.*

[3] *D. Scaradozzi, L. Cesaretti, L. Screpanti and E. Mangina, "Identification and Assessment of Educational Experiences: Utilizing Data Mining With Robotics," in IEEE Robotics & Automation Magazine, vol. 28, no. 4, pp. 103-113, Dec. 2021, doi: 10.1109/MRA.2021.3108942.*

[4] *TALKING BLOCKS.*
https://www.univpm.it/Entra/Ricerca/Universita_e_Impresa/PoC_FASTER/Programma_FASTER/Progetto_TALKING_BLOCKS_-_FASTER .
Last accessed 08/07/2022.

[5] *Jormanainen and E. Sutinen, "Using Data Mining to Support Teacher's Intervention in a Robotics Class," 2012 IEEE Fourth International Conference On Digital Game And Intelligent Toy Enhanced Learning, 2012, pp. 39-46, doi: 10.1109/DIGITEL.2012.14.*

[6] *P. Blikstein, M. Worsley, C. Piech, M. Sahami, S. Cooper, and D. Koller, "Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming," Journal of the Learning Sciences, vol. 23, no. 4, pp. 561-599, 2014.*

[7] *P. Y. Chao, "Exploring students' computational practice, design and performance of problem-solving through a visual programming environment," Computers & Education, vol. 95, pp. 202-215, 2016.*

[8] *D. Scaradozzi, L. Cesaretti, L. Screpanti and E. Mangina, "Identification of the Students Learning Process During Education Robotics Activities", Frontiers in Robotics and AI, DOI:10.3389/frobt.2020.00021*

[9] *L. Cichella, "Study and Development of "Talking" Educational Tools", 2021. Available at http://hdl.handle.net/20.500.12075/7436*

[10] *R. N. Gulesin, "Educational Robotic Tools for the Identification and Modelling of Learning", 2021. Available at http://hdl.handle.net/20.500.12075/7439*

[11] *L. Cesaretti, "How students solve problems during Educational Robotics activities: identification and real-time measurement of problem-solving patterns", 2020. Available at http://hdl.handle.net/11566/274358.*

[12] *Scikit-learn - Wikipedia.*
https://en.wikipedia.org/wiki/Scikit-learn .
Last accessed 09/07/22.

[13] *Google Colab.*
https://research.google.com/colaboratory/faq.html .
Last accessed 09/07/22.

[14] *Garbage collection - Wikipedia.*
https://en.wikipedia.org/wiki/Garbage_collection_(computer_science) .
Last accessed 09/07/22.

[15] *Packaging Python Projects.*
https://packaging.python.org/en/latest/tutorials/packaging-projects/ .
Last accessed 09/07/22.

[16] *GDK - Wikipedia.*
https://en.wikipedia.org/wiki/GDK .
Last accessed 09/07/22.

[17] *GTK Scene Graph Kit - Wikipedia.*
https://en.wikipedia.org/wiki/GTK_Scene_Graph_Kit .
Last accessed 09/07/22.

[18] *Manifests - Flatpak documentation.*
https://docs.flatpak.org/en/latest/manifests.html .
Last accessed 09/07/22.

[19] Minka, T. P.. “Automatic choice of dimensionality for PCA”. In NIPS, pp. 598-604

[20] Tipping, M. E., and Bishop, C. M. (1999). “Probabilistic principal component analysis”. Journal of the Royal Statistical Society: Series B (Statistical Methodology), 61(3), 611-622.

[21] UIFlow.
https://flow.m5stack.com/ .
Last accessed 09/07/22.

[22] Discord.
https://discord.com/ .
Last accessed 09/07/22.

List of Figures