Technical University of Crete
School of Electrical and Computer Engineering

A platform to benchmark inference algorithms based on sensor network data

by **Athanasios Rentzepopoulos**

A thesis submitted in partial fulfillment of the requirements for the diploma degree of
Electrical and Computer Engineering

October, 2022

A platform to benchmark inference algorithms based on sensor network data

ABSTRACT

This work offers a means of accessing a database with sensor network (e.g. soil moisture) measurements/data for use with inference algorithms. The design problem is approached by creating a web application based on microservices hosting a user interface (UI), a scalable back-end that runs the algorithms, and database storage. Algorithms are expected to be Python scripts developed offline and tested either offline or online. This multi-modal operation poses a compatibility concern. We create a library that exclusively handles data input and output for the scripts that import it, with different implementations depending on the context of the user scripts. We also explored an efficient way to execute scripts, in the back-end of the web-app, allowing for robust interruption of running scripts and parallel execution. This service also includes a queueing platform that handles the input and output of the user scripts and takes care of graceful process start up and shutdown. The user experience is also accounted for, with responsive web UI and source code analysis to automatically find out the number and type of input streams. Finally, we test the system by utilizing the library to parse and pre-process data and implement an inference algorithm to run on the pre-processed data.

Acknowledgements

First, I would like to thank Professor Aggelos Bletsas for assigning the topic, continuous guidance, advice and constructive criticism throughout the preparation of the thesis. I would also like to thank the members of my committee Associate Professor Vasilis Samoladas and Professor George Karystinos for evaluating my work. The greatest gratitude I ow to my closest friends, especially Vasilis Papageorgiou and Spyros Pepas, who stood by me through thick and thin. I also thank, Vaggelis Karatarakis for his help regarding inference algorithms. Finally, I sincerely thank my family for their support throughout my studies, and their patience, especially during the difficult times.

Contents

Chapter 1

Introduction

Processing of sensor data is at the heart of the Internet of Things (IoT) era. Sensor data is stored near the sensor network whereas data processing is often performed in different locations. Bringing sensor network data closer to the centers of processing requires careful design to ensure data integrity and context maintenance. In parallel, the evolution of Artificial Intelligence (AI) requires that the performance of inference algorithms that are being developed can be evaluated based on the same data that are accessed following a consistent method.

In this project, a tool that facilitates both the above problems was designed, developed, and tested. This tool is a platform that accepts arbitrary code, which may implement inference algorithms, data manipulation, reporting, or any other data processing activity. This code then is provided with access to sensor network data stored independently. The platform ensures correct data serialization/de-serialization and allows process execution either in a local or a client-server configuration. The platform is web-based. The web application is designed using microservices [1] that operate in a Docker [2] environment. There are three service layers: a front end, the execution service, and the database service following a classic three-tier application design. The front end layer uses Flask [3]; the execution layer is written in Python [4] and utilizes several Python standard libraries to provide input, output and data processing tools to the user scripts. Finally, the storage layer uses MongoDB [5] that provides document-based access which is suitable for this type of operation.

An important part of the platform functionality is devoted in data manipulation, which requires a uniform interface regardless of the implementation mode (local use, as part of the execution service, or as part of the front-end service). The execution service includes a queueing platform that handles input and output of the user scripts and takes care of graceful process start up and shutdown. Finally, the front-end service takes care of communication between the user and the execution service, as well as analyzing scripts and data.

1.1 Prior Art

This thesis's problem overlaps with existing tools, commercial and otherwise. One of these tools, called RapidMiner [6], uses a Graphical User Interface to combine existing algorithms visually. One of the most popular tools for this topic is Google Colab [7]. It is broadly used across different research areas and is a hosted Jupyter notebook service providing access to cloud computing resources. Similar–Jupyter notebook based–tools are available from Amazon [8] and Microsoft [9]. However, the goal of this thesis was purely educational and to cater to the specific use case of accessing remote data more easily and executing code near the data.

1.2 Thesis Outline

This thesis is structured as follows: After this short introduction, the web application structure and design is presented in Chapter 2. Chapter 3 is devoted to the data manipulation features of the platform. In Chapter 4 we present the heart of the platform, i.e., the execution service and in Chapter 5 the frontend service. Chapter 6 we present a demonstration of the platform using two typical user scripts, one performing data manipulation activities and the second implementing an Loopy Belief Propagation (LBP) inference algorithm on moisture datasets created by the first script. Finally, Chapter 7 presents conclusions and suggestions for future work.

Chapter 2

Web APP

The concept of this project is rooted on the problem of having data useful for research purposes, but no way to access it. In this case the data is stored on a remote database and the means of processing it is through Python scripts. The approach selected was to create a web application that stores data, datasets and scripts in a database and offers a cohesive user interface where “members” can run said scripts, whereas “visitors” can only view the results. This architecture allows for the project to be extensible in terms of both functionality and visitor’s experience.

2.1 Structure

For the infrastructure of the project a microservices [1] architecture, and more specifically Docker, was chosen over a monolithic one. The actual structure, as shown in Fig.2.1, includes a service[1] responsible for serving the content to the browser (Chapter 5), a service for executing the Python scripts (Chapter 4) and a few containers[2] for the database service for the system (Section 2.2). Communication in this project is handled by docker’s networking system which allows for *networks* to be created and containers to connect to them. Each *network* can be marked as internal or external which dictates whether it is connected to the outside world or not. The *networks* used are, one external for the users to connect to the frontend service and one internal for all services to communicate over, protected from the outside world.

Docker also provides ways for passing vital information like credentials and configurations to the services.

- Non-sensitive information like paths, durations, connections Uniform Resource Identifiers (URIs), encryption algorithms etc. is passed through environmental variables. This information is stored in a plain text configuration file and Docker passes it to the specified container(s) in the form of global variables at the Operating System (OS) level. This configuration file is subject to version control.

- Sensitive information (secrets) is passed using encrypted files. Docker encrypts secrets before passing them to their respective containers. These files are not subject

[1]Service refers to the functionality that is provided to the rest of the project. A service can be deployed as a number of containers.

[2]Container refers to the actual Virtual Machine and the code hosted that implements the functionality of a service.

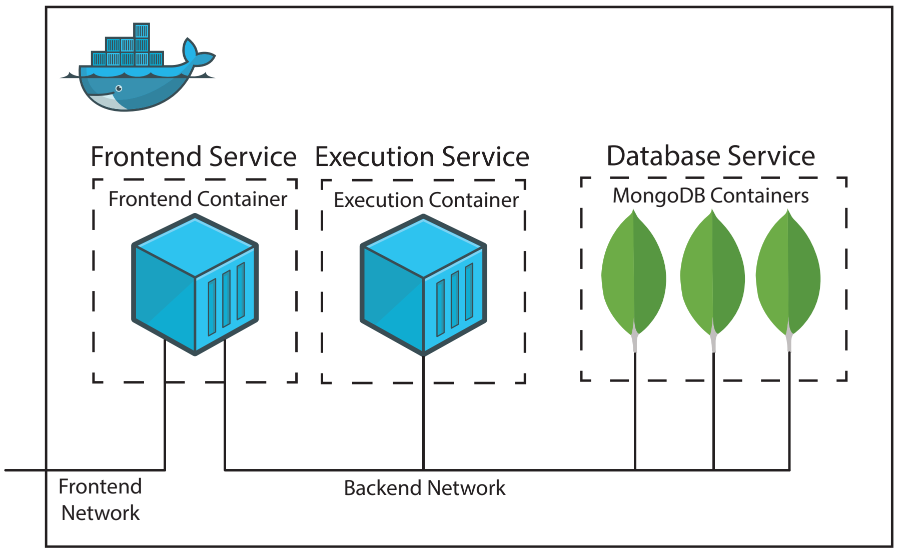


Figure 2.1: Docker services structure.

to version control since they describe information like passwords and encryption keys that should be generated anew by the operator upon system deployment.

Finally, Docker provides persistent storage using the volumes feature. Docker volumes are used by the databases as their physical layer. They are also used during development to allow updating the code without the need for the respective container to be restarted.

2.2 Database

For the database of this project MongoDB was chosen for its NoSQL, document-oriented approach and for opting in for *Consistency* and *Partition tolerance* on the CAP theorem [10]. The CAP theorem states that any distributed data store can provide only two of the following three guarantees:

- **C**onsistency: Every read receives the most recent write or an error.
- **A**vailability: Every request receives a (non-error) response, without the guarantee that it contains the most recent write.
- **P**artition tolerance: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

To circumvent any *Availability* shortcomings, it offers a feature called *Replication*.

MongoDB organizes data in *databases*, *collections* and *documents* and requires authentication per individual *database*. Table 2.1 presents the rough correspondence to SQL terms. In MongoDB, a *document* is a file that uses notation similar to JSON

Table 2.1: MongoDB terminology

SQL	MongoDB
Database	Database
Table	Collection
Row	Document

(JavaScript Object Notation)[3]. During initialization, a JavaScript program is run to set the *databases*, their credentials, their *collections* and any indexes needed.

For the needs of this project the nine *collections* created are organized in four *databases* based on the data each holds (see Table 2.2). There is a *database* for moisture data that isn't organized into datasets (retrieved by the remote database), one for data relating to scripts (scripts, datasets, and other files like figures), one for the execution queue (see Chapter 4) and one for the users' credentials. Each service has access only to the *databases* necessary based on the *collections* it needs. This arrangement adds a layer of security in case a container gets compromised since each database has separate credentials.

It should be noted that all *collections* of the Processed Data Storage *database* handle file-like data and not data that is just tabular-compliant. This alone would not be a problem for a document based database, but said files can be too large to efficiently store in one document. So for these files a MongoDB feature called GridFS is utilized, which splits the actual data part of the document into chunks, while also allowing storing metadata.

Table 2.2: The project's *databases* and *collections*

Database	**Collection**	**description**
User Storage	Users	Users' information
	Refresh Tokens	Used to refresh the JSON Web Tokens
Raw Data Storage	General Info	State of other collections in this database
	Sensor Info	Details for each sensor
	Data	Soil moisture readings
Processed Data Storage	Datasets	Datasets used as inputs for scripts
	General Data	Other data (used for file type outputs of scripts)
	Scripts	Scripts and metadata
Hot Data Storage	Execution Queue	Priority queue for script execution

[3]MongoDB uses a file structure called BSON which is an augmented version of JSON that allows for Binary data. JSON (JavaScript Object Notation) is an associative container, wherein a string key is mapped to a value.

2.3 Main programming language

The choice of the programming language to use for the user code was based on ease of use, especially in relation to linear algebra calculations. The most popular choices are Python and Matlab[4]; however, the latter is part of a proprietary ecosystem which is comparatively harder to run and and interface with. For the code base of the back-end server, there was a much greater variety of languages and frameworks. Since Python was chosen for the user's code, it made sense to use it also for the back-end to facilitate processing of data into datasets[5] (see Chapter 3). It should be noted that using Python was not a strict requirement since other languages could just run simple Python scripts to handle data and datasets, but choosing Python for the back-end means that everything can be done in one language.

2.3.1 Libraries used

Python has a selection of frameworks to develop web applications with, the most popular of which are Django [11], Flask [3] and FastAPI [12] . For this project, Flask was chosen because it is lightweight, easy to develop for and has strong community support. It is used in both the frontend service (to serve the user) and the execution service (to allow communication over HTTP with the frontend service). Front-end development used the Jinja2 templating engine, which allows to enrich HTML files with data from the calling code before serving it to the user. Each service that utilizes Flask has a main module that is called at the beginning which imports the main package and starts the Flask app. The main package is responsible for creating and describing the functionality of the Flask app and has modules and sub-packages for each part of the functionality.

Apart from Flask related libraries, the following were also instrumental for this project:

- `pymongo` was used to connect to the main database of the system.
- `pyodbc` was used to access the remote database that hosts the original sensor readings.
- `numpy` used to handle and process data since it is widely used for linear algebra calculations.
- `matplotlib` was used as the plotting tool the scripts use to produce results.
- `typeguard` was used to enforce type checking–which Python lacks–in key parts of the code, especially when interfacing with user generated scripts.

2.3.2 Development tools created

In order for the code to interface with the database a subpackage in the main package of each service was created to handle communication. For this, a class approach was

[4]To some extend R can also be used, however, it is more commonly used for statistical analysis.

[5]The datasets are serialized objects containing the data which allows them to be directly deserialized and used in the scripts.

selected for the benefits of inheritance. The abstract base class is responsible for handling credentials, connecting to the correct *database* (utilizing the `pymongo` library), actually creating a connection instance, and creating a wrapper around communication methods that ensures atomicity of transactions. So, for each *collection* only one class with the necessary logic is created.

Apart from that, each service has a module with debugging tools and one with general tools. The latter has functions for reading and checking files (for secrets) and globals (for environmental variables), creating UUIDs[6], and handling the different representations of time (time object, integer and human readable string).

[6]UUID stands for Universally Unique IDentifier and is used in every ID of anything that is stored in the database.

Chapter 3

Data manipulation

At its core, the project is responsible for the flow of data from the means of storage to the executing script and vice versa. The data used by the script is stored as a byte sequence, while the metadata containing the type and characteristics are more complicated. This is because the greater use that metadata provides to the user, the author of the script, and the dataset library since it represents the constraints that might pose an incompatibility. More specifically the metadata is saved as an object of a class that stores the type of the dataset and any specific constraints based on that type. For example one of the supported types is an *image list*, which includes metadata regarding the list length and the image dimensions. This information is represented internally as key-value pairs, which MongoDB supports intrinsically (Section 2.2). However, if the dataset is saved on to a file, then the metadata is first converted to a byte stream and combined with the data.

3.1 Dataset

The scripts that are compatible with the project are written in Python and are expected to process multidimensional arrays using a library called `numpy`. This library is used to perform mathematical operations and is optimized at a low level (utilizing the ability to execute computationally expensive tasks in C rather than Python), thus making it industry standard for its performance. The sensor data that is used for the inference algorithms refers to soil moisture readings and is organized using `numpy`'s `ndarray` data type, which is a multidimensional array. Additionally, some algorithms use data from images to augment their calculations with additional information. These images can be represented also as `ndarrays`, but if the images differ in size for any reason then they cannot be combined. This creates a need for an additional data structure utilizing Python's list class (which is more commonly known as *arrays* in other languages) and `numpy`'s `ndarray`, storing each image as an `ndarray` entry in the list.

Apart from the classes used to handle data, there may be a need to configure script parameters. For example, one might need to test only part of the code for a demo and not run parts that create statistics and debug information; this can be managed by a configuration variable passed on to the script. Finally, any other type of data not accounted for should also be able to be passed to the script, but marked as such. All of these classes of data, as seen on Table 3.1, represent different dataset types in the dataset library (Section 3.2) and serve as inputs and outputs for the scripts.

Table 3.1: Types of datasets

Class	**Data type**	**Additional information**
`numpy` array	`ndarray`	Dimensions of the array
Image list	List of `ndarrays`	Length of list and (opt.) dimensions of array
Command	String	Type to convert it to
Unspecified	Any	No extra information

These datasets, as mentioned, can function both as inputs and as outputs of scripts, but some scripts may require a more visual type of output. More specifically, a lot of the scripts produce results in the form of graphs, plots and other visual elements that are not covered by the aforementioned datasets. To tackle this issue, a new type of output was created which is assumed to be a figure file and is saved in its final, presentable form, both when stored as a file and in the database. This means that the resulting file can be viewed as an image, depending on the file type it is saved as. To facilitate the creation of said figure files, a Python library called `matplotlib` is used which, like `numpy`, is also widely used in this field. The only notable difference to datasets is that the figure must be prepared by the script before being saved by the dataset library.

3.2 Dataset library

In order for the script to use the data it needs to load it, which normally would be as easy as:

```python
with open('path\to\file') as f:
    data = f.read()
```

However, when the code runs on the server, retrieving data is not as simple. The programmer would need to know how to retrieve it, what database is used, all the paths to the dataset and the security details that should never be known in the first place. So the programmer would need to create a different implementation for running the code on the server which would be complicated, error prone, and insecure.

To overcome this hurdle, a package was created (referred to as *dataset library* in this text), which is imported by the user's script and handles fetching incoming and storing outgoing data to the respective storage medium. This package abstracts the difference in implementation behind the scenes and, in case it runs on the server, it handles interfacing with the database without exposing the credentials. The differences between the different implementations are explored in Section 3.3.

3.2.1 Structure

The basic premise of the dataset library is that it should be extendable without needing refactoring, while the basic functionality of retrieving and storing data should be easy to alter. To achieve this, a class approach was selected in order to take advantage of inheritance and modified with extension classes to augment the behavior. More specifically, for each dataset type (including the figure file type) and for each valid operation for it (refer to Table 3.2), a class was created that inherits from the respective abstract base class, Reader or Writer, and any extension classes in may need (Table 3.3). For example the reader class for image lists inherits from the *reader base* class and the *list* and *nparray* extension classes. In turn the reader and writer base classes along with all the extension classes all inherit from the same base class.

Table 3.2: Operations permitted per data structure

Data type	**Data structure**	**Read**	**Write**
Dataset	`numpy` array	✓	✓
	Image list	✓	✓
	Command	✓	✓
	Unspecified	✓	✓
File	Figure		✓

Table 3.3: Extension classes each dataset type always inherits from

Dataset type	**Extension classes**		
	`nparray`	List	Command
`numpy` array	✓		
Image list	✓	✓	
Command			✓
Unspecified			

The base class is responsible for saving the class data type and defining the default *check* method that checks that the type of the data–read or to be written–is compatible with what is expected. Inheriting from the base class, the reader and writer base classes define the *read* and *write* methods, respectively, that interface with the file system or database and call the *check* method as needed. The extension classes define extra member variables that are used in the overloaded version of *check*, which verifies compatibility of the dataset's metadata. After all the utility classes are created, the final classes are just a combination of a base class and any necessary extension classes as noted in Table 3.2. Finally, functions corresponding to each utility class are created that wrap the process of creating an object, calling the *read* or *write* method, and disposing of it. The final structure can be seen in Figure 3.1.

3.3 Variations

The library's main purpose is to handle the data so that the script does not have to, regardless of the environment. For this reason, even though the library's interface with the script is consistent, the underlying functionality has differences in implementation so

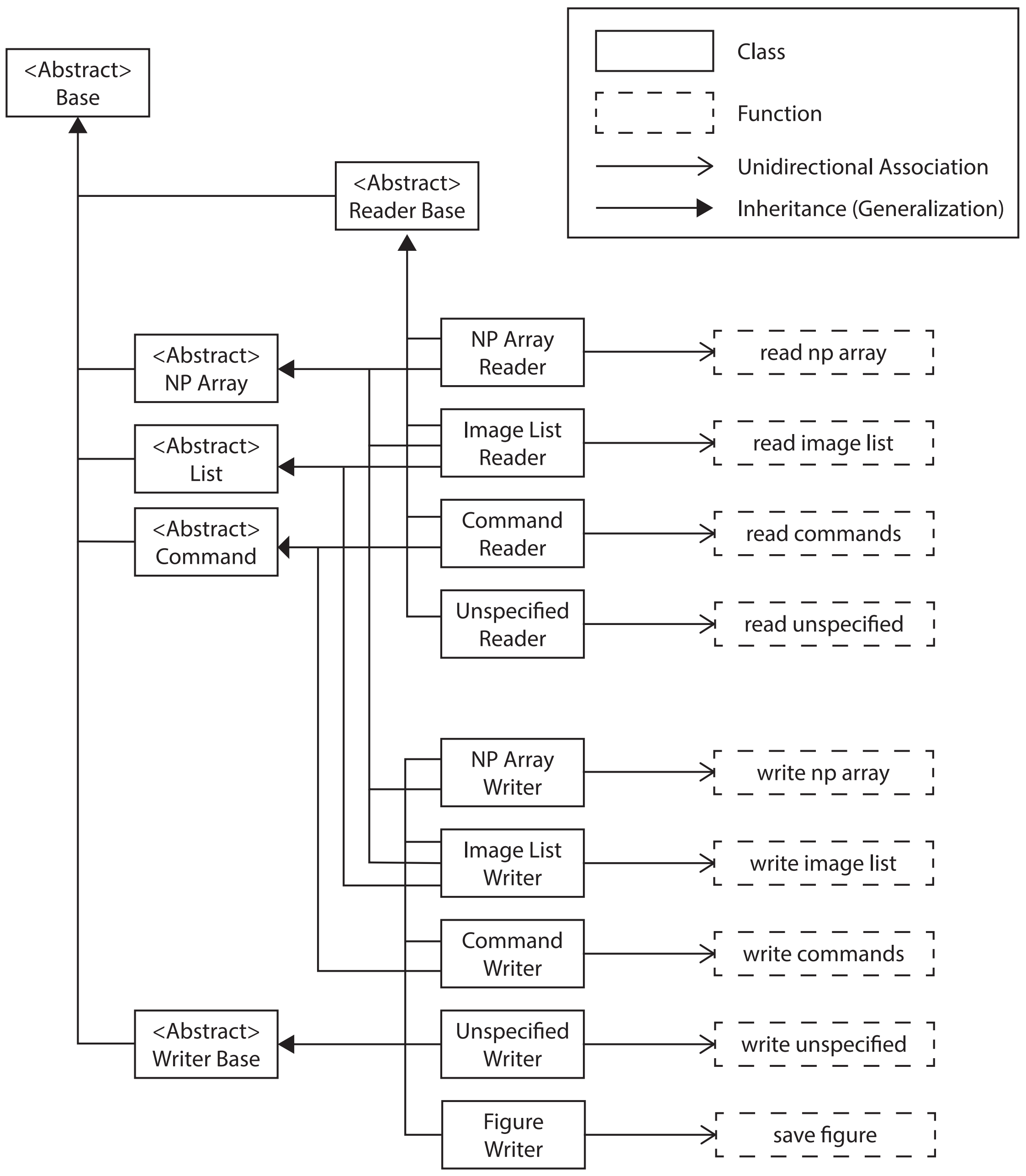


Figure 3.1: Package's class structure.

as to adapt to the respective environment. The variations of the library depend on the type of the caller, which means there is a variation for local use, one for the use in the *execution* service (see Chapter 4) and one for use in the *frontend* service (see Chapter 5).

The only assumption made about the library is about how the script imports it.

Python has different ways to import a package depending on a combination of whether or not the code is split into different files, where the library is located and preference. However, when the the code is contained in one file (as is the expected use case in this project) there is only one acceptable import method for packages that are not installed using the package manager; the package's code must be located on the same folder as the script. Then the code to import it is as follows:

```python
import dataseting_lib
```

Snippet 3.1: Example of absolute import

This type of import, and its equivalents, are referred to as absolute imports and their difference from their counterparts, relative imports, is the way Python names the imported package's internal components. Assuming a development of a package `a` in Python that uses a module or package `b` and `a` has modules that require `b`. The absolute method is to place `b` in the directory that the main module (which calls the package `a`) is placed. Now, all modules of `a` can import `b` in the same way as in snippet 3.1. With this import method all internal components of `b` will be named as descendants of the `main` module. For the relative import method, `b` should be placed inside the package `a` (or even one of the sub-packages). Modules of `a` that require `b` would need to import it knowing their relative location to `b` (see snippet 3.2). With this import method all internal components of `b` will be named relatively to its location compared to the main module. So, if `b` is placed at the top level of package `a` then `b` will be named as descendant of `a`.

```python
# In this case the package is place in the parent directory of this
    ↪ module
from .. import dataseting_lib
```

Snippet 3.2: Example of relative import

The importance of the import method and Python's naming mechanisms is realized when using the module `pickle`. This module is used to serialize and deserialize Python objects and uses said naming mechanisms in the process. `pickle` is used in this project to serialize objects of dataset classes in order to transmit them between the different means of accessing them (locally, in the execution service etc.). This allows a dataset in file format to be uploaded to the website and be deserialized and understood. The only requirement is for Python to assign the correct names to the dataset in both environments, which can be guaranteed with absolute imports. In essence, as long as all variations of the dataset library use the same serialization parameters (same name of package and module, same serialization library and same import method), then they can all parse files from one another.

3.3.1 For local use

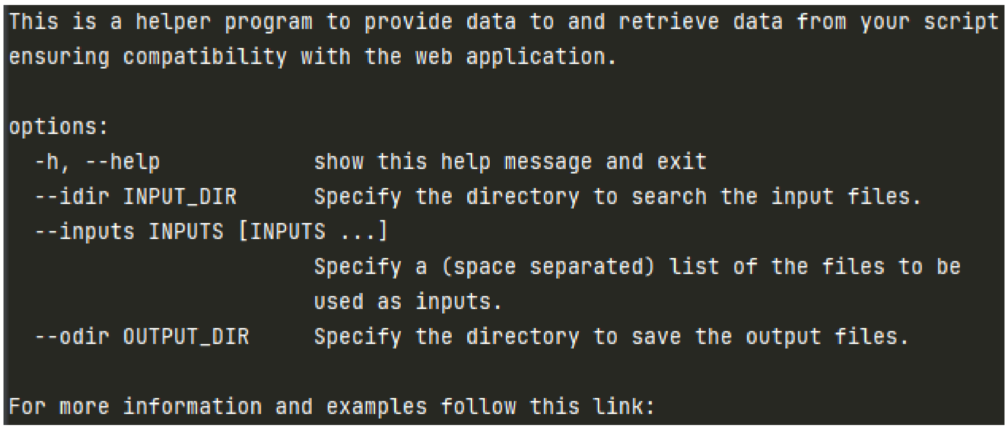
The variation of the library created to run in a user's local machine treats datasets as files. Since the library does not have access to the database, it only serves the purpose of helping the programmer to test their code before uploading it to run on the server.

So the implementation of reading and writing simply translates to reading and writing to disk files, while the only non-trivial part is saving the metadata. However, like data, metadata is also serializable, which means it can be stored in the same file.

The most notable detail of the local variation is the function that parses the console arguments when calling the script. The reason of its existence is because of how the script is called in the server and preserving consistency. More specifically the user should not specify how to search for the file when writing the code since this information becomes irrelevant when executed at the server. The solution is to pass the input files as arguments when executing the script, parse it using a function provided by the dataset library (`parse_console_arguments`) and use these files when read operations are requested.

```
python -m test_script --idir . --inputs test_dataset.ds
```

Snippet 3.3: Calling a script that uses the `parse_console_arguments` function.



```
This is a helper program to provide data to and retrieve data from your script
ensuring compatibility with the web application.

options:
  -h, --help            show this help message and exit
  --idir INPUT_DIR      Specify the directory to search the input files.
  --inputs INPUTS [INPUTS ...]
                        Specify a (space separated) list of the files to be
                        used as inputs.
  --odir OUTPUT_DIR     Specify the directory to save the output files.

For more information and examples follow this link:
```

Figure 3.2: Help message for assigning datasets to a script.

3.3.2 For use in the execution service

The execution service (see Chapter 4) is responsible for running user uploaded scripts, which require the dataset library. When a script is about to be executed it is retrieved from the database, written to a file and placed in a directory with the dataset library. However, since data is saved in a database (namely MongoDB, Section 2.2) instead of the file system, a sub-module was created to handle interfacing with the database and handling any sensitive information. In this way the read and write methods of the respective classes make simple calls to this sub-module.

3.3.3 For use in the frontend service

Apart from user generated scripts, this library is also used in the *frontend* service of the web app. One of the most important uses of the *frontend* service is creating datasets from raw data provided by the user of the site, e.g. compiling uploaded images into an *image list* dataset. For these operations the writer functions of the library are used, adjusted to utilize the connection to the database created for the frontend service (see Section 2.2). Note that, since the structure of the code base of the *frontend* service is compartmentalized into a package imported by the main module (see Chapter 5), the position of the dataset library in the folder structure is on the same level as the main module This means that any sub-module of the main package that requires it must uses absolute importing. Lastly, the dataset library is used in the code analyzer sub-module of the frontend service for the tokenizer to work properly (see Sec.5.1).

Chapter 4

Execution service

One of the main services of this project is the *execution* service which handles running user-uploaded scripts. This functionality is created as a separate service with its own HTTP-based interface so that its resource needs are addressed directly. This allows to allocate more CPU power to its container(s), compared to a service like the frontend, if there is a need for that. This decision also facilitates better behavior of the whole system in edge cases of downtime like maintenance or unexpected crashes of the *execution* module, since the only unavailable part of the system is the *execution* service, while every other works as intended (see Section 2.1).

The endpoints of this service are managed by Flask and are the basic actions passed on to the *execution* module. These actions are:

- **Add script to queue**
  Inputs: script's UUID, inputs' UUIDs

- **Stop script**
  Inputs: script's UUID

- **Execute next *task* in queue**
  Inputs: None

In general, the *execution* module will automatically pick up the next *task* in queue to complete, but in certain cases (such as in case there are multiple containers for this service) a need to manually pick up the next *task* may arise.

4.1 Queue system

The *execution* service implements a First In First Out (FIFO) type of queue for the scripts it needs to execute. Since MongoDB supports ACID transactions[1], utilizing it to implement the queue removes the hustle of managing race conditions between multiple *consumers* and *producers*.

When a script is selected to be executed, an entry in the database is created (managed by the *execution* module) representing a *task*. For this *task* the information saved is

[1]ACID stands for Atomicity, Consistency, Isolation, Durability and ensures operations (like insert, update or delete) meet certain criteria. In this case *producers* and *consumers* or the queue cannot race for their respective operations.

the script to be executed (represented by its UUID), the list of inputs for the script (also represented by their UUIDs), the list of outputs of the script (populated after the execution), the state of the *task* (like pending, running etc), and some additional information that depend on features (a feature may be to include anything written to `stdout`). When the module is available to execute a script, it looks for the oldest entry in the database in state *pending* and changes it to *executing*.

4.2 Execution module

The backbone of the *execution* service is the *execution* module, responsible for managing the *task* queue, managing the processes and everything regarding running the scripts. The basic structure is a frontier class, a manager process and a number of children processes as depicted in Figure 4.1.

The frontier class serves the purpose of creating the manager process and creating an interface for the rest of the service to communicate with said manager through a custom messaging system. This system is built on top of Python's queue which is implemented

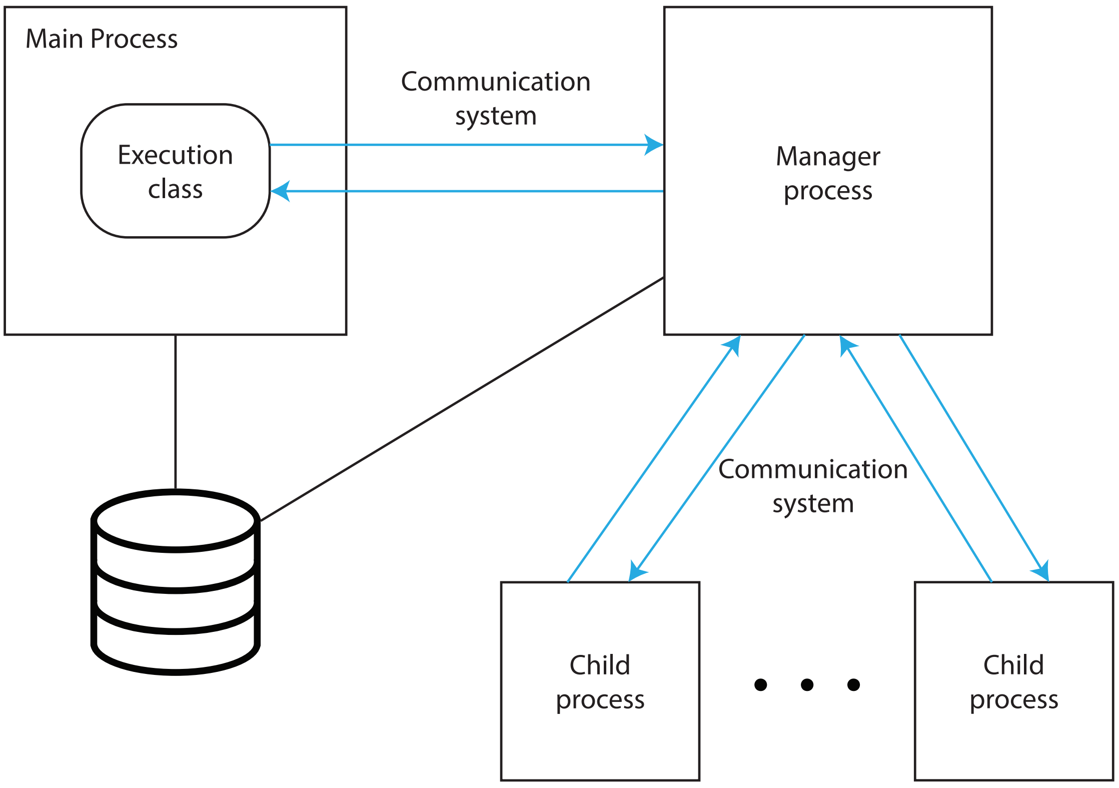


Figure 4.1: Structure of the *execution* module.

to be multiprocessing-safe and uses a condition variable[2] (cv) to ensure atomicity. It is also used as a communication path between the manager process and its children processes, but with separate queues and condition variables. The structure for this module may appear complicated at first (considering that running an external program is fairly trivial), but this complexity is necessary.

The main objectives of the *execution* service is to be able to run scripts, support parallelization, stop any script if requested and be responsive while scripts run. Simply running a script is fairly trivial in Python by using the *subprocess* module:

```python
from subprocess import Popen

process = Popen('path/to/script.py')
process.communicate()
```

However, calling the method *communicate* of the *process* object blocks the normal execution of the main (and only) thread making the whole service effectively unresponsive until the process has finished and does not leave any option to stop the execution if needed.

To achieve responsiveness, running the script (using `Popen` from the *subprocess* module) was moved to a separate process (the children processes) whose main function is to monitor the execution and act based on the commands passed on to it. However, the issue of blocking the main thread of the process remains, so instead of blocking on the `communicate` method, the routine[3] polls the process to check whether it has finished or not. This allows the child process to also monitor the messaging system between it and the manager and process any commands it receives while also notifying the manager if the script finished running.

Moving the execution of the script into a separate process enables the option to create multiple child processes to achieve our goal of parallelism without adding more complexity. The only problem is managing the child processes and specifically listening to the messages they generate and acting on them while also being responsive to any incoming commands (typically coming from the HTTP endpoints through the proxy class). To solve this issue, a manager process was created whose main purpose is to coordinate the messages from the children and from the main process.

The final structure uses a minimum of two generated processes to handle the duties of controlled execution of Python scripts, but there are alternative approaches that accomplish the main objectives:

- The first approach involves leveraging the microservices based structure of the project, to create more containers instead of more processes. This approach is valid, but by default consumes more resources per child spawned since a container is a (very stripped down) Linux based virtual machine.

[2]Condition variables are synchronization primitives that enable threads to wait until a particular condition occurs.

[3]In this case *routine* refers to a piece of code that is not necessarily bound to a language structure like a function, class or module.

- Another approach would be using a production environment to run Flask that supports multiple processes like *Gunicorn*; however, this creates a dependency to the production environment that may cause problems in the future when extending the feature set. Using this approach, would also introduce complications in managing the processes, making sure they are not underutilized while handling HTTP requests, and being able to cancel the execution of the correct script.

- Lastly, it should be noted that Python has two more ways to achieve parallelization of *tasks*; using *threads* and the *asyncio* module. However, both of these options do not actually utilize the system's resources to run on multiple CPU threads and instead run using a single one[4].

4.3 Manager process

The manager *process* was introduced to pass commands from the main process to the child processes without blocking the main *process* on conditional variables (e.g. waiting for possible responses from said children). Apart from the main objective, however, it also handles creating and killing child *processes*, consuming the execution queue when there are idle *processes* and other similar responsibilities. This Section goes into detail on how this *process* works and what it does.

To spawn child *processes* in Python, the most streamlined way is to use a module called `multiprocessing` which handles creating *processes*, abstracting OS-level operations. Using `multiprocessing` module entails creating a function that runs on the newly created *process* and passing a reference to it as an argument to the `Process` class, along with any arguments the function would need. As hinted in snippet 4.1, the module provides a number of synchronization primitives that enable communication between the different *processes* in an easy (and *pythonic*[5] for that matter) way[6]. The primitives used in this project are;

- The `Condition`, which is the condition variable processes block to when they need access to a resource.

- The `Queue`, which is the primary type of resource used to transmit inter-process messages.

- The `event`, which acts as a Boolean flag.

The convention for accessing resources is that each process has a condition variable for synchronization and a queue from which it only consumes (similar to a *consumer* in a

[4]In most languages, threads provide true parallelism, but in C-based Python (the one used in this project and the most popular implementation of Python) the threading module uses the Global Interpreter Lock and forces every thread to use it, effectively running in a single thread.

[5]*Pythonic* often refers to a coding style that leverages Python's features to produce code that is clear, concise and maintainable.

[6]Note that the *threading* and *asyncio* modules also have synchronization primitives, but each module can only use its own.

```python
import multiprocessing as mp

def example_process(process_cv, process_queue, a_number):
    with process_cv:
        # Wait on the condition variable until the queue is not empty.
        process_cv.wait_for(lambda: not process_queue.empty())

        # print something using the item from the queue and one of the
    ↪ arguments
        print(f'Question: {process_queue.get()}, Answer: {a_number}')
    return

cv = mp.Condition()
q = mp.Queue()
a_number = 42

process = mp.Process(target=example_process,
                     args=(cv, q, a_number))
process.start()
with cv:
    q.put('What is the answer to life the universe and everything')
    cv.notify_all()

# Prints: "Question: the answer to life the universe and everything,
    ↪ Answer 42"
```

Snippet 4.1: Creating a process with the *multiprocessing* module.

pipe). For a process `A` to transmit a message to process `B`, it should gain control of the condition variable of process `B` and then write to the queue of `process B`.

The function for the manager process has five arguments that are populated when the process spawns. Two of the arguments are used for the manager's condition variable and queue, two more for the response, and the last one to set the default number of child processes. Inside the function there is number of other functions, for the various *tasks* the manager encounters, along with some variables that are effectively global in the sense that the aforementioned functions have access to them since they are defined in the same context. Among these variables there are some responsible for storing information about the processes, their state and efficiently searching them, two variables to help regulate the number of child processes and some objects to communicate with the database (using the service's database interface sub-package).

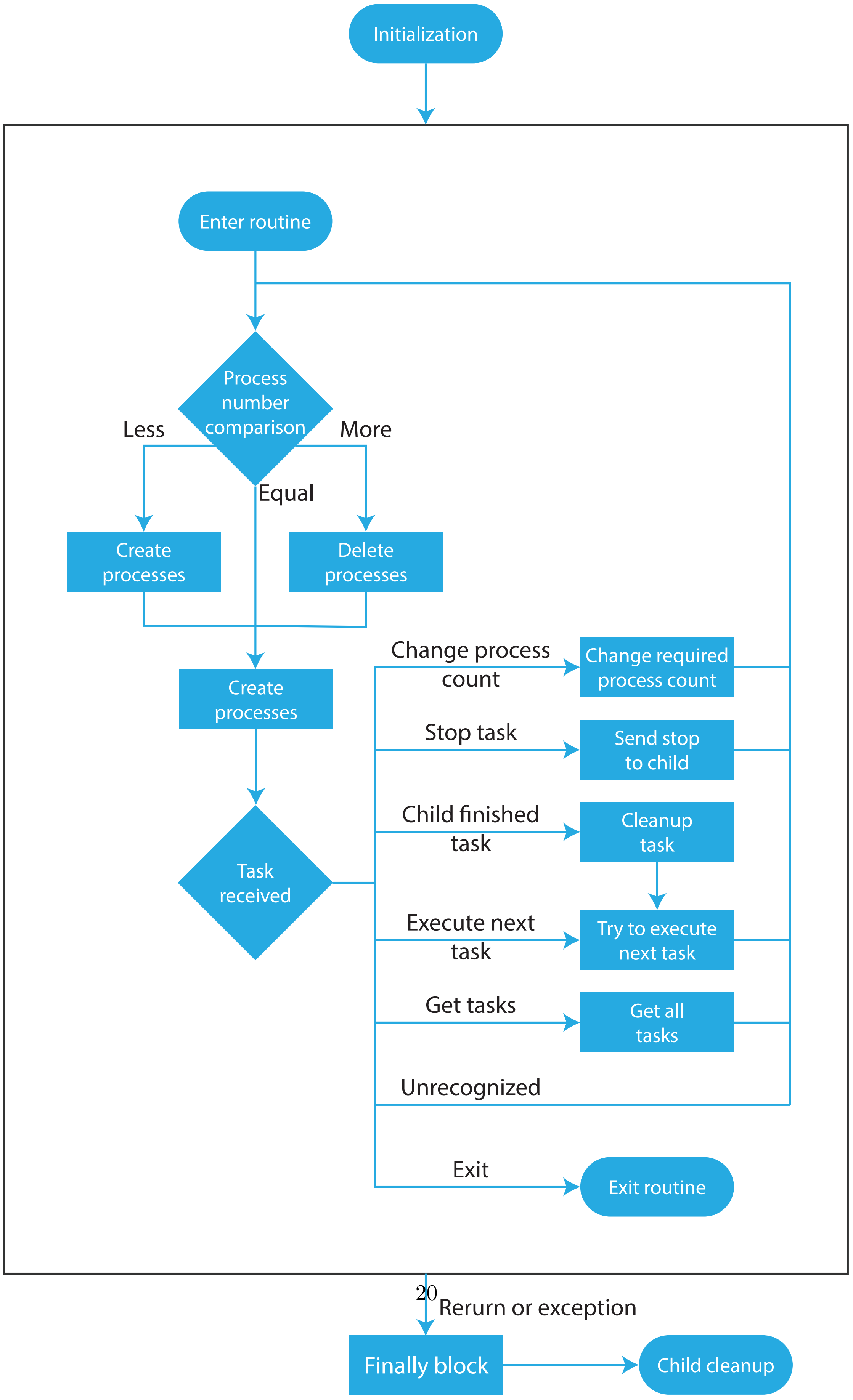


Figure 4.2: Manager process flowchart.

The most used function inside the manager function is `send_response` which is responsible for sending a Python dictionary, accepted as input, as a message to the main process. These messages are sent as responses to actions inside the manager and even though are very useful during development, most of them are ignored by the other end (the main process's proxy class). `send_response`'s implementation acquires the condition variable, writes to the queue, and notifies the main process (similar to lines 19 through 22 in snippet 4.1). It should be noted that this function is also implemented in the child processes to transmit messages to the manager, with the only difference being that the manager actually listens to and acts upon them.

The main routine of the process is encapsulated in a try-finally block[7] ensuring that if any unaccounted for exceptions or deliberate premature returns occur, then the child processes will be cleaned up properly, along with any remnants of them (like the folders assigned to said processes). The actual routine running inside the try block is an infinite loop consisting of two parts; the former is responsible for the number of child processes, while the latter handles any messages in the manager's queue.

The reasons for handling the child processes inside the main loop is to be able to change their number on the fly and in the case a child dies to be able to spawn a new one. The first step in this part is to clean up any folders (in the designated directory for child processes' folders) that are not recognized belonging to its children, which helps with cleaning up of processes that died unexpectedly. Then the number of children is compared to the expected and the manager creates or destroys them depending on said comparison. If there is a need to reduce the number of children, then the routine attempts to remove and clean up processes marked as available. After that the number of processes still remaining to be removed is saved in a variable to note how many of the already occupied processes should be terminated when finished with their respective *tasks*. If there is a need to create processes, then, for each child that needs to be added, the routine attempts to create it for up to a maximum number of attempts. Reaching that number without successfully spawning a child, prematurely ends the manager process (calls return), triggering the finally block before exiting.

When the manager process creates a child it must also create the resources it will use. The first resource is the directory the process will use which should be isolated from other processes' directories to improve stability. This directory is created as a sub-directory of a predefined root folder and then the dataset library is copied to it. This is done because the script is placed in that directory when it is about to be executed by the specific child and needs access to the library to function. After the directory becomes available to the process, the synchronization primitives are created, used to communicate with the process. Apart from the condition variable and the queue, whose functionality is already explained, an event[8] is also created which, when set, signifies that the child has finished initializing. Following that, the actual process object is created using as target a function

[7]A try-finally block refers to normal execution of code occurring in the try block (a block of code that begins with the try keyword) and in case of exiting that block (either intentionally or due to an exception) calling the finally block.

[8]Events are a synchronization primitive in Python that have only two states: set and not-set. At creation time they are not set.

describing the child's behavior (see Section 4.4) and an argument list comprised of the aforementioned synchronization primitives, the manager's condition variable and queue, and the path to the child's directory. Afterwards, the process is marked as daemon[9], forcing it to quit if the manager dies, and gets started. Then, the routine acquires the child's condition variable and waits for up to 1 second for the child to finish initialization and set the event, which exists to prevent a race condition between the child initializing and the manager assigning a *task* to it. If the initialization is not completed within that time, then the child is scrapped and, after cleaning up, the routine moves to the next attempt. The final step is to save this process's information, mark it as available and break out of the attempt loop (which signifies successful creation to the routine).

Aside from creating child processes, the manager also needs a way to destroy them, the functionality of which is split into two functions. The first one is tasked with simply making sure the child dies by any means necessary, but gracefully if possible. To do that, the function first sends a command to the child that it should exit and then attempts to join the process for up to 2 seconds. If the process has not exited by that time, then a *kill* signal is sent to it and stops monitoring its state. This function is used only when the child process fails the initialization step or in the other function, which handles cleaning up every trace of the child process altogether. The second function accepts as input only the PID of the child to be terminated and infers the rest of the information, like the child's condition variable, queue, process object and folder path, from the manager's pseudo-global variables. After that, it calls the previous function to kill the process, removes any reference to it from the manager's variables, updates the database if the process was amidst execution, and deletes the directory assigned to it.

After the child processes are managed, the routine acquires the manager's condition variable and waits until the queue receives a message which is a command the manager should execute (assuming it recognizes it). These commands are expected to come from either the main process or the child processes. The simplest commands are changing the number of processes, getting the state of the child processes, and forcing the manager to exit gracefully, all of which are sent by the main process. One more command recognized by the main process is stopping a specific *task* which entails retrieving the child executing it from the *ticket*'s UUID[10] (i.e. retrieve the *task* ID), retrieving the child's communication endpoint and sending a `stop` command to it. The final command the main process can send is to initiate searching for *tasks* to execute in case the child processes are underutilized, which is handled by a function. This function takes a process marked as available (if there are any) and the next *ticket* from the database (if there are any), validates the existence of the script and the inputs in their respective databases described in the *ticket*, saves the script in the child's designated directory as a file (previously saved as raw data in the database), sends a command to the child to execute the program along with the list of inputs, and marks the process as busy. An important point to make is that only the manager process is communicating with the

[9]The term daemon in the context of Python's `multiprocessing` module means that if the parent process dies, then the child dies as well. This ensures that the unexpected exits don't leave orphan processes.

[10]The terms *ticket* and *task* are used interchangeably and denote a scheduled execution of a script.

database, so the transition from byte array in the database to executable Python script is handled by the manager and not the child.

Child processes send messages that present the result of the script execution attempt. This result may be:

- Successful execution.
- Error occurred during execution.
- Error occurred during setup.
- Script execution was canceled.

All of these are handled by a single function, which removes the child from the busy processes, updates the state of the ticket and any errors that may have occurred, and then either kills the child or marks it as available depending on the number of leftover processes to remove from the first part of the main routine. As previously stated the child processes do not communicate with the database so the step of updating the ticket is handled by the manager.

4.4 Child processes

The child process is a lot simpler in nature than the manager process since its only two objectives are to listen for commands from the manager and handle the execution of scripts (see Figure 4.3). To do both at the same time, the process polls every 1 second for updates from either source, instead of blocking in perpetuity, handles the situation and then informs the manager. As previously stated, the child process has access to its own and to its manager's condition variables and queues as well as an event which by default is not set. The first steps the function describing the process goes through are retrieving its PID, changing its directory to the designated one, and setting the event provided to it, thus signaling to the manager that the initialization step is completed. As with the manager process there is a dedicated function to send messages to the manager, a main infinite loop where all the non-initialization code is placed, and the aforementioned loop is encapsulated in a try-finally block which handles clean-up in the case of uncaught exceptions or expected premature returns.

Inside the main infinite loop, the routine acquires the process's condition variable and blocks on it for periods of 1 second until either the script has finished executing or the queue is not empty. If the script has finished executing then the routine sends the appropriate response to the manager and cleans up the resources used. Then, any messages from the manager are handled, the only noteworthy of which are those about starting or stopping the execution of a script. To initiate execution, the manager needs to send the UUID of the *ticket*, the input list and the module name (i.e. the name of the file the code was placed in after it was retrieved by the database), along with the actual command. After that, the files absorbing anything written in `stdout` and `stderr` are created which are then passed on to the `Popen` class of the `subprocess` module,

```python
from subprocess import Popen
import sys

stdout_ = open('path/to/stdout.txt' 'w')
stderr_ = open('path/to/stderr.txt' 'w')

sub_process = Popen(
    [sys.executable. '-m', module_name, ticket_id, *input_list],
    stdout=stdout_,
    stderr=stderr_)
```

Snippet 4.2: Creation of a subprocess to run a script.

along with the executable details (see snippet 4.2). If any exception may occur, then the execution is concluded and the appropriate error response is passed on to the manager. It should be noted that even though the file handlers for the files of `stdout` and `stderr` are not closed in the snippet 4.2, they are in the actual implementation when the program finishes (or is forced to finish) execution. Lastly, if the manager or the finally block request the premature completion of the script, then the appropriate signal is sent to the process, using the `subproces`'s object, any resources used are cleaned up and then a response is sent to the manager.

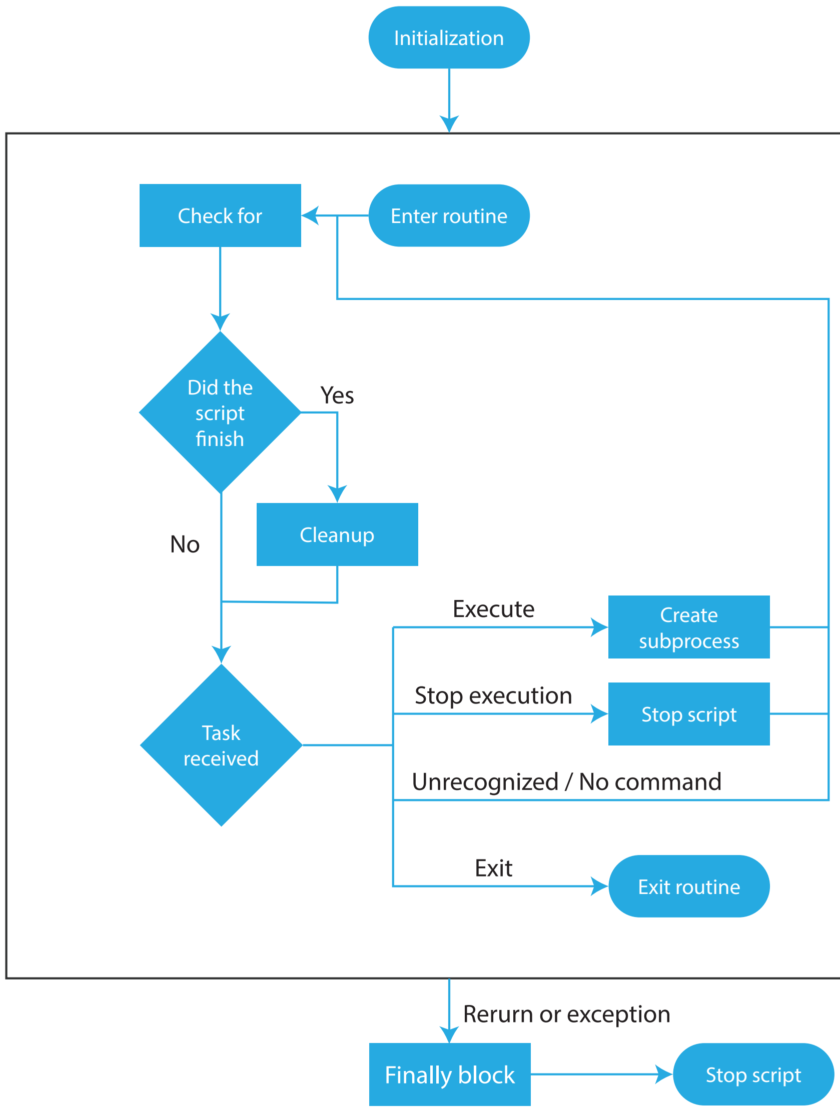


Figure 4.3: Child process flowchart.

Chapter 5

Frontend service

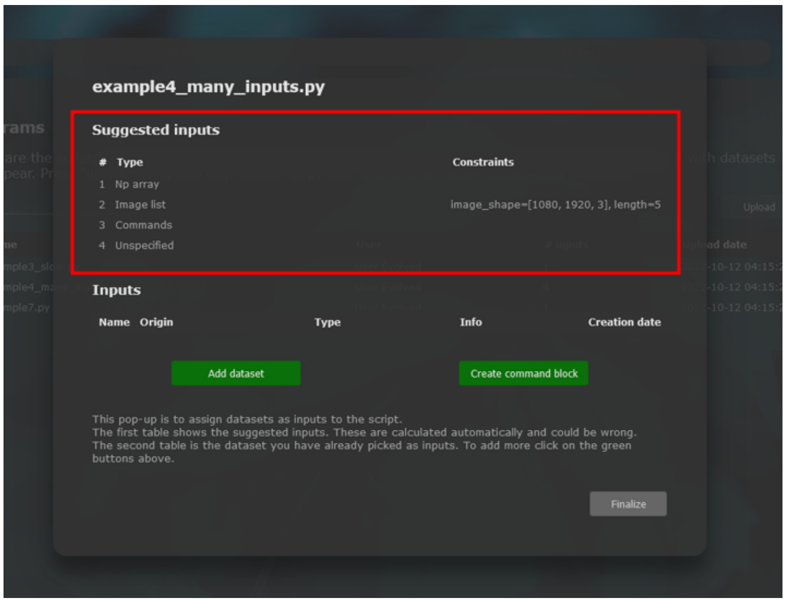
The frontend service refers to the code running on the server responsible for communicating with the browser and doing any work required before either responding or communicating with the local database, the remote database or the execution service. More specifically, the duties of this service include:

- Communicating with the remote database to retrieve data.
- Processing data into datasets.
- Saving datasets.
- Tokenizing and saving scripts.
- Sending commands to the execution service.
- Serving the pages to the browser.
- Responding to the AJAX endpoints (that ususally require metadata from the database).

5.1 Code analyzer module

The dataset library organizes data in types incompatible with one another. This means that, when running a script, a user must already know the input dataset types and their order of appearance, or it will fail before it even gets to process the data. To circumvent this issue, a module was created that inspects the script when it is uploaded, detects calls to reader (and writer) functions, and saves the findings as part of the script metadata. This is used to create suggestions of the inputs for the user to know what datasets to pair it with before executing it (see Figure 5.1).

To extract the information from the script a Python module was used called `ast` which stands for Abstract Syntax Trees. This module provides tools for creating a traversable tree based on the input and the current *abstract grammar* [13]. This tree consists of nodes of predetermined types, as shown in the example snippet 5.1, where the simple `print('hello world')` command is analyzed. To use the module effectively a *visitor* class needs be created defining methods called on specific nodes. For example, if there was a need to detect calls to the `print` function, then a method called `visit_Call` would



example4_many_inputs.py

Suggested inputs

#	Type	Constraints
1	Np array	
2	Image list	image_shape=[1080, 1920, 3], length=5
3	Commands	
4	Unspecified	

Inputs

Name	Origin	Type	Info	Creation date

This pop-up is to assign datasets as inputs to the script.
The first table shows the suggested inputs. These are calculated automatically and could be wrong.
The second table is the dataset you have already picked as inputs. To add more click on the green buttons above.

Figure 5.1: The menu for running a script with suggestion for the inputs.

need to be created that detects that the `func` field is a node of type `Name` with fields `id='print'` and `ctx=Load()`.

```python
code = "print('hello world!')"

import ast
print(ast.dump(ast.parse(code), indent=4))

# prints:
# Module(
#    body=[
#        Expr(
#            value=Call(
#                func=Name(id='print', ctx=Load()),
#                args=[
#                    Constant(value='hello world!')],
#                keywords=[]))],
#    type_ignores=[])
```

Snippet 5.1: Analyzing code with Abstract Syntax Trees.

For the purposes of this project the only node types useful were `Import`, `ImportFrom` and `Call`. `Call` is used to detect calls to the reader and writer functions and the import node types to detect how the calls are formatted. More specifically depending on how the dataset library was imported there `Call` node has different inner structure because it has different *grammar* (see snippet 5.2).

```python
# === Package based imports ===

# Only importing the pakcage
import a_package
a_package.a_module.a_function()

# === Module based imports ===

# Simple module import
from a_package import a_module
a_module.a_function()

# Importing the module with alias
from a_package import a_module as my_module
my_module.a_function()

# Problematic, but valid importing of the module
import a_package.a_module
a_package.a_module.a_function()

# Also importing the module with alias
import a_package.a_module as my_module
my_module.a_function()

# === Function based imports ===

# Simple and specific
from a_package.a_module import a_function
a_function()

# Simple, but general (not recommended)
from a_package.a_module import *
a_function()

# Alias for function name
from a_package.a_module import a_function as my_func
my_func()
```

Snippet 5.2: Different ways to call the same function based on import method.

5.2 Flask and Jinja 2

As mentioned in Section 2.3.1, Flask is the framework of choice used with its default templating processor to enrich the HTML pages served, Jinja2. Developing for this framework requires creating an object of the `Flask` class and applying certain properties to it, like adding a login manager and defining the secret key used for form-related security reasons. After that, the endpoints are defined and the application is started.

The structure of the main package is modeled after one of the recommendations. It consists of a modules for Flask functionality, along with folders for the Jinja2 templates and the static files (like images, CSS and JavaScript). The most notable module for

Flask of the main package is `views.py` in which the endpoints are described. The main package also contains code unrelated to the framework, these being the code analyzer module, the remote database connection module, and the local database connection sub-package.

5.2.1 Endpoints

An endpoint refers to a link that the *frontend* service makes accessible to others that can communicate with it. These endpoints are described in the file `views.py` which is part of the main package. The Flask application object provides a decorator[1] that creates an association of the specified URL to a function describing the code that gets executed before the response is calculated. In snippet 5.3 for example, when the user types the Uniform Resource Locator (URL) `www.this_site.com/example_page` (or gets redirected to it), then the function `endpoint_example_page` is called which in turn renders the template `example_page.html`. In this project there are endpoints for the pages a user can access and for URLs that are meant to be reached by the JavaScript code running on the browser using the AJAX[2] pattern. The latter is data that is requested after the page loads that would be too expensive to send at once or depends on the actions of the user.

```
from . import app

@app.route('/example_page')
def endpoint_example_page():
    return render_template('templates/example_page.html')
```

Snippet 5.3: Example of creating an endpoint using flask.

5.2.2 Forms

Pages that require input from the user, like sign-up, are most commonly implemented using HTML forms. Flask provides tools assisting the creation of pages that use HTML forms and processing resulting data with the module `wtforms`. With this module it is possible to create a class whose class variables are entries in the form with the desired constraints described by *validators*. These are classes that express a limitation the field should have. For example, in snippet 5.4 the `username` is a field of type `string` that is required and has a specific length.

[1]By definition, a decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it. In this case when the URL in the decorator is called then Flask calls the function decorated with it.

[2]AJAX stands for Asynchronous JavaScript and XML. It is a set of web development techniques that uses various web technologies on the client-side to create asynchronous web applications. Ajax, web applications can send and retrieve data from a server asynchronously (in the background) without interfering with the display and behavior of the existing page.

```python
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField, BooleanField

class LoginForm(FlaskForm):
    username = StringField('Username',
        validators=[DataRequired(), Length(min=2, max=30)])
    password = PasswordField('Password', validators=[DataRequired()])
    remember_me = BooleanField('Remember me')
    submit = SubmitField('Log in')
```

Snippet 5.4: Example of a log-in form class.

Classes describing forms can be used in the endpoint functions to create either the HTML code when combined with the Jinja2 template, or retrieve the data of the form and validate it against the constraints. Using Flask forms simplifies the backend code that checks the form, the frontend code by using the much more laconic Jinja2 template syntax, and security since it is handled entirely by Flask[3].

5.2.3 Login

To manage access to pages based on whether the user is logged in or not, Flask's login manager is used. This system abstracts restricted access behind a decorator to the functions of endpoints that require it. However, there is some setup required in the form of a class that represents a logged in user with member variables any useful information the endpoints may need to access. Then the endpoint function that handles logging-in creates an instance of this class and passes it on to the login manager so that, when a restricted page is requested by that user, it enables them to access it.

5.3 HTML, CSS and JS

The pages of the site are written in HTML enriched with Jinja2 for the markup, CSS for the styling and JavaScript for the client-side logic. There are seven pages:

- Home
- About
- Sign-up
- Log-in
- Results
- Datasets

[3]Flask provides the option of handling the security ascpect of forms by simply using a hidden tag in the template, provided that the `SECRET_KEY` configuration variable is set.

- Run

Out of these, *Datasets* and *Run* are restricted since they provide enough control to the user to severely hinder the experience for other users. The *Results* page shows completed runs along with any files produced, allowing to view and download said files. *Datasets* is a composite page that entails all aspects that can be associated with them. In it there is a sub-page to view existing datasets and upload a new one (created using the local variant of the dataset library–see Section 3.3), one to view the soil moisture data and create a dataset out of them, and one to help create a dataset out of images uploaded. The *Run* page is also composite and has a sub-page with all the uploaded scripts allowing to upload a new one or simply select one and pair it with datasets to add to the execution queue (see Chapter 4). It also has a sup-page with every entry in the execution queue along with their status allowing to stop any tickets *running* or *pending*.

As previously mentioned JavaScript is part of the frontend stack used to dynamically change the content of the site. One of the most notable uses of it is AJAX, which is used to load content from the server after the page has finished loading. This, can allow for larger data to be loaded after the base page has, allowing for a more responsive experience. In some cases it can also prevent content that is not required from being loaded, thus reducing the amount of data sent to the user

Chapter 6

Demos

In order to test the functionality of this project two scripts are created, each testing a different aspect. These scripts simulate a real use case of this thesis by implementing an inference algorithm and running it on data from a sensor network. The full code for the following examples can be found on appendix A.

6.1 Example 1: Creating a dataset from raw data

The first script handles preprocessing a moisture dataset from NASA into a usable table which simulates creating a dataset and uploading it to the website for use by the uploaded scripts. The original dataset is parsed into Python using the library netCDF4 and the result is a *masked array*, a type of NumPy array. Additionally the area the original dataset represents is an extensive region of Europe meaning that the grid of sensors is quite large. This means that the two operations needed to be completed by the preprocessing script is to select a smaller area to work with and convert the data into a compatible NumPy array. After that, the dataset library becomes useful for saving the resulting dataset into a file to be uploaded to the website. As seen in snippet A.1 the code for that is quite simple.

6.2 Example 2: Running LBP on a moisture dataset

The second script uses the previous dataset, runs an inference algorithm, and plots its performance as a result. The problem tackled in this script is inferring the value of sensors when they occasionally do not function. The algorithm used is called Loopy Belief Propagation (LBP) and combines historical data from the, at times, malfunctioning sensor and their neighbors to create an estimation of their values that have the highest probability.

LBP works by propagating beliefs between neighboring nodes based on distributions for each node which, in this case, are estimated using previous readings of the sensors. More specifically when a node i passes a message (i.e. propagates a belief) to node j, the calculation of said message involves the distribution of node i, called self potential, the correlation of the two nodes, called edge potential, and the messages that i has already "received". The formula for message passing can be seen in Eq. 6.1 in which ϕ denotes the self potential, ψ the edge potential and $N(i)$ the neighbors of i.

$$m_{i \to j}(x_j) = \sum_{x_i} \phi_i(x_i)\psi_{ij}(x_i, x_j) \prod_{k \in N(i) \setminus \{j\}} m_{k \to i}(x_i) \tag{6.1}$$

This calculation is done for each message possible from any node to any of its neighbors and for each state (x_i) said neighbor can be in.

If the set of states all nodes can take is $\boldsymbol{S} = \{s_1, \ldots, s_n\}$ and there are n total states, then the Eq. 6.1 can be vectorized as follows:

$$\boldsymbol{m}_{i \to j} = \boldsymbol{\psi}_{ij}\boldsymbol{\phi}_i \bigodot_{k \in N(i) \setminus \{j\}} \boldsymbol{m}_{k \to i} \tag{6.2}$$

where $\odot$ denotes element-wise multiplication. By this approach every message $\boldsymbol{m}_{i \to j}$ and self potential $\boldsymbol{\phi}_i$ are vectors of length n whose value of states that are impossible is 0. In the same way $\boldsymbol{\psi}_{ij}$ is a n by n, zero-padded matrix. This changes are necessary for more efficient calculations when implementing the algorithm. These messages are passed multiple times, hence the "loopy" in the name, and are used at the final step to calculate the belief for each node–Eq. 6.3.

$$p_{\mathsf{x}_i}(x_i) \propto \phi_i(x_i) \prod_{j \in N(i)} m_{j \to i}(x_i) \tag{6.3}$$

This can also be vectorized in the same way as in Eq. 6.1.

$$\boldsymbol{p}_{\mathsf{x}_i} \propto \boldsymbol{\phi}_i \bigodot_{j \in N(i)} \boldsymbol{m}_{j \to i} \tag{6.4}$$

The script for this example (snippet A.2) reads the data, creates the state set, creates the self and edge potentials, runs LBP for different percentages of broken sensors, calculates the error metrics for each test, plots the results, and saves them to a figure. The self potential for each node is calculated as the distribution observed by the samples of states. The edge potential of a pair of nodes is the joint distribution of the respective nodes and, in order to calculate it, the IPF algorithm is used [14, 15].

For this example the graph's structure is assumed to be a lattice which means that messages are passed to four directions on each node. With this assumption edge nodes accept messages from virtual non-existent nodes that transmit a message that does not affect the outcome of the calculations. In the same manner the outgoing messages of the edge nodes to them are ignored. This means that the 3-dimensional tensors holding the messages for each direction all have the same size and, with careful planning, can be combined in parallel, which is the source of the optimization.

Calculating an outgoing message towards a direction requires the incoming messages from the same direction and the two from the perpendicular orientation as shown in Figure 6.1. This means that for a m by n lattice the effective messages for the two horizontal directions are m by $n-1$ and for the vertical $m-1$ by n. However, if these messages are saved in matrices and their inward-most row or column is padded

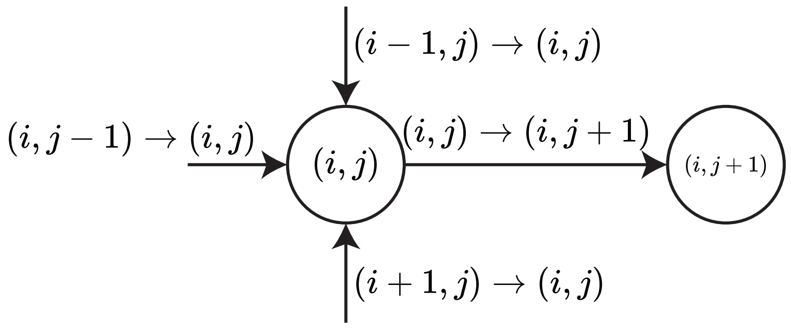
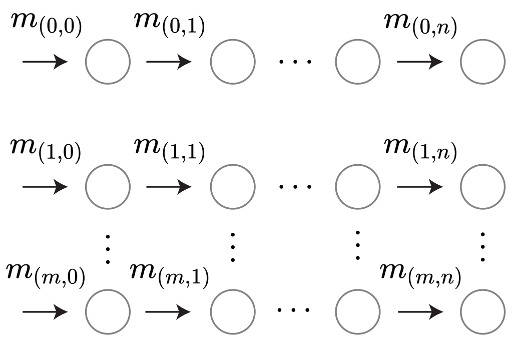


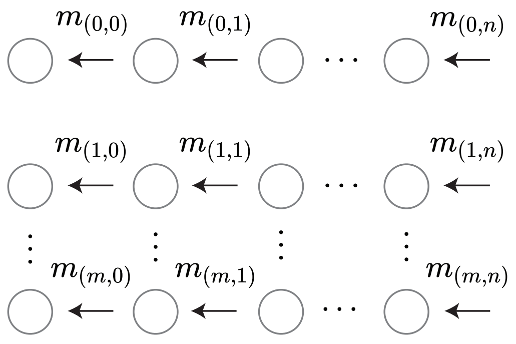
Figure 6.1: Messages required for belief propagation.

with neutral messages, as shown in Figure 6.2, then these matrices all have the same m by n dimensions. Moreover, the node in position (i,j) accepts messages from the position (i,j) of each message matrix, which means that calculating the beliefs can be parallelized by element wise matrix multiplication (similar to the Eq. 6.4). It should be noted that messages are stored in 3-dimensional tensors instead of matrices because for each message the information stored is the likelihood of each state.

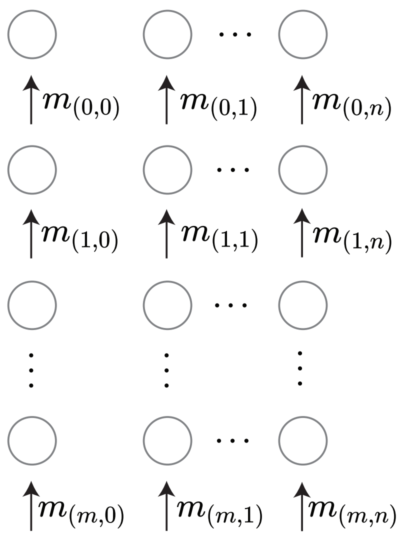
In a similar fashion, calculating the messages for the LBP can also be parallelized by combining the messages the transmitting node receives. So a message transmitted from node (i,j) to $(i,j+1)$ requires the messages from the directions *up*, *down* and *right* for the node (i,j), which are in the (i,j) position of the matrices. Again, referring to the structures that hold the messages as matrices is done to convey that their whole state is used, which is a vector. It should be noted that only the effective messages are calculated, meaning that the *left* message from node $(0,0)$ is not calculated since there is no use doing so.



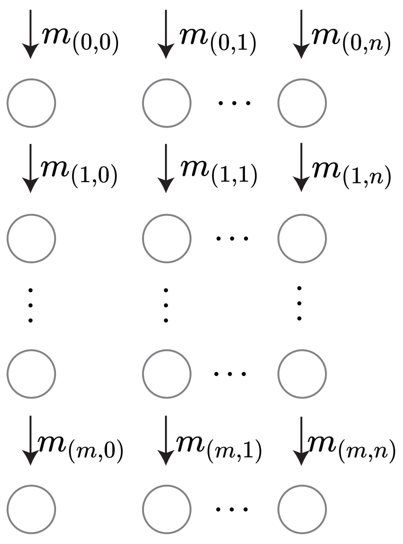
(a) Messages towards right.



(b) Messages towards left.



(c) Messages towards up.



(d) Messages towards down.

Figure 6.2: All messages passed.

Chapter 7

Conclusions & future work

We designed, developed, and tested a platform that can be used to benchmark inference algorithms working on sensor network data. The platform operates as a standard web application with separate UI, logic and storage tiers. The platform advantages include scalability, ease of deployment in different IT environments and clean separation between the algorithms' code and sensor data. A robust metadata handling process ensures consistent operation of the user scripts for local use and as part of the execution service of the platform.

There are several aspects of the platform that could be improved. First and foremost, the security of the platform is basic and in many cases to facilitate development we designed aspects of the platform considering that the provided scripts are not malicious. Another improvement would be to compact data saved on the database, since in the current form little effort has gone into avoiding redundances. Additionally, the entirety of the systems are created without utilizing unit testing which, for the scale of the project, is quite important.

Finally, some thoughts about future work include:

- Automation to support batch operations: to carry out several benchmarks a scripting engine could be used to facilitate the repeatable execution of test script suites.
- Decentralized design: When sensor network data are in the TB range, it may make more sense to implement the platform as a distributed system, where, instead of copying the data from their initial location to the place where the script lives, we send the script to be executed where the data is and send back only the script output.
- Black-box abstraction layer: One of the greatest augmentations to the website portion of this project would be to create an environment allowing for users to directly experiment with different algorithms. This would allow for appeal to a greater audience and demonstrate in real time some of the most interesting algorithms.
- Integration of the implementation of various inference algorithms to the python library, which can be used for rapid prototyping and research purposes.

Appendix A

Demos' code reference

This appendix is just for the code referenced by 6.

A.1 Example 1 code

```python
# Import the dataset library
from dataseting_lib import writers, parse_console_arguments
# Import the parser for the data
import netCDF4 as nc

parse_console_arguments()

file_path = './EU_ESSMRA_daily_ensmean_CLM-PDAF_3Km_v1.200001.nc'

# Read the data from the file
dataset = nc.Dataset(file_path)
humidity_data_variable = dataset['H2OSOI']

# Pick a subset of it
humidity_data_np_masked = humidity_data_variable[:, :100, :100]

# Convert data into a usable values
humidity_data_np_masked_normalized = humidity_data_np_masked * 100
humidity_data_np_array = humidity_data_np_masked_normalized.filled(
    ↪ fill_value=1)

# Write data as dataset
writers.write_np_array(humidity_data_np_array, 'humidity1.ds',
    ↪ if_file_exists='overwrite')
```

Snippet A.1: Create an *np-array* dataset from data.

A.2 Example 2 code

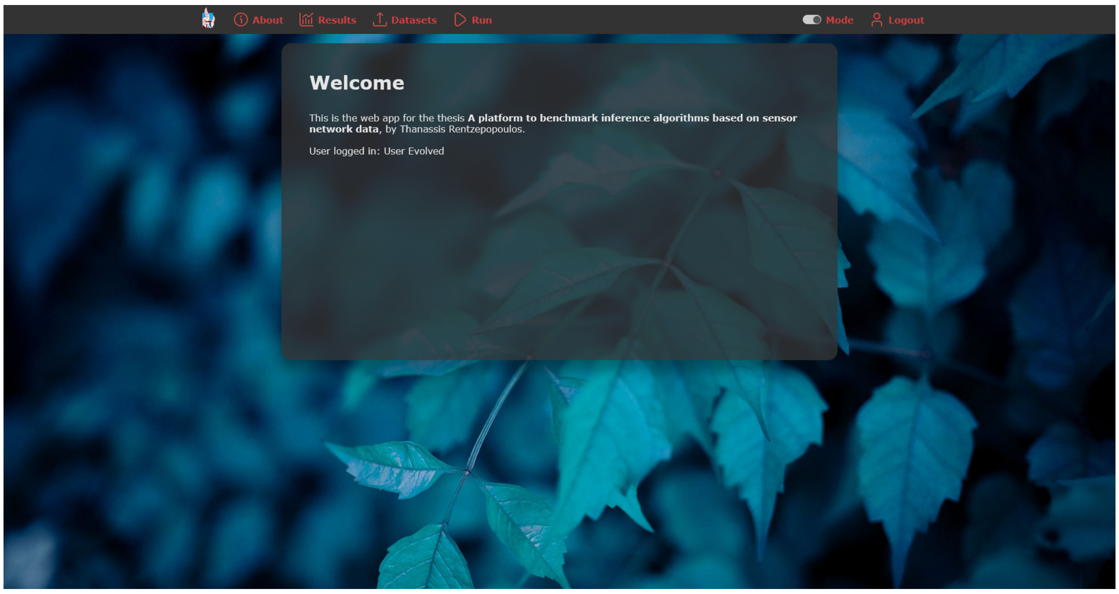
```python
# Optional imports
from time import perf_counter
from typing import TypeVar

# Imports for computation and results
import numpy as np
import matplotlib.pyplot as plt
# For type checking
from typeguard import check_argument_types

# For reading and writing datasets
from dataseting_lib import readers, writers, parse_console_arguments



# Globals
num_iters_lbp = 10
eps = 1e-8

wsn_size_1 = 100
wsn_size_2 = 99
numberOfFiles = 6

T = TypeVar('T')

# Functions for part 1

def get_files_and_samples(soil_moist_data: np.ndarray):
    """Reads the dataset and picks the train and test data."""

    # Train data
    samples = [np.ceil(soil_moist_data[i, :wsn_size_1, :wsn_size_2]) for
    ↪ i in range(numberOfFiles)]

    # Test data
    beliefs_opt = np.ceil(soil_moist_data[numberOfFiles, :wsn_size_1, :
    ↪ wsn_size_2])

    return samples, beliefs_opt


def build_the_lattice_graph(samples: list[np.ndarray]):
    """Rearranges data into a one numpy array."""

    moisture = np.zeros((wsn_size_1, wsn_size_2, len(samples),))

    for i in range(numberOfFiles):
        s = samples[i]
        for j in range(s.shape[0]):
            for k in range(s.shape[1]):
                moisture[j, k, i] = samples[i][j, k]

    return moisture


def get_classes(moist: np.ndarray):
    """Returns the unique values and the number of them."""
    classes = np.unique(moist)
    count = classes.shape[0]
    return classes, count


def get_f_prob(unique_values, num_of_values, moisture):
    """Creates the self potentials for each sensor."""

    # Initialization
    f_prob = 1e-8 * np.ones((wsn_size_1, wsn_size_2, num_of_values))

    # Lookup table for each value of the "unique_values" variable
    uv_dict = {
        value: index for index, value in enumerate(unique_values.tolist()
    ↪ )
    }

    for i in range(wsn_size_1):
        for j in range(wsn_size_2):
            values = moisture[i, j]
            uniques = np.unique(values)
            percents = np.histogram(values, bins=np.append(uniques, np.
    ↪ inf))[0] / numberOfFiles

            # Overwrite the default value with the percentage
            for v, p in zip(uniques, percents):
                f_prob[i, j, uv_dict[v]] = p
            pass
        pass
    return f_prob


# Functions for part 2

def ipf(T: np.ndarray[T, T],
        R: np.ndarray[T, 1],
        C: np.ndarray[1, T],
        num_of_values, epsilon=1e-3, max_it=100):
    """Iterative Proportional Fitting. Used in edge potential calculation
    ↪ ."""
    check_argument_types()

    n = num_of_values
    m = num_of_values

    total_error = np.sum(np.abs(np.sum(T, 1) - R)) + np.sum(np.abs(np.sum
    ↪ (T, 0) - C))

    S = np.ones((n, m))
    errors = []
    for count in range(max_it):
        errors.append(total_error)
        if total_error <= epsilon:
            break

        for i in range(n):
            S[i, :] = S[i, :] * R[i] / np.sum(S[i, :])

        for j in range(m):
            S[:, j] = S[:, j] * C[j] / np.sum(S[:, j])

        total_error = np.sum(np.abs(np.sum(S, 1) - R)) + np.sum(np.abs(np.sum(S, 0) - C))
        pass
    return S


def get_edge_potentials(f_prob, num_of_values):
    """Calculate the edge potentials."""

    # Initialization matrix for IPF.
    T = np.ones((num_of_values, num_of_values))

    # Horizontal edge potentials
    horizontal_psi = np.zeros((wsn_size_1, wsn_size_2 - 1, num_of_values, num_of_values))
    for i in range(wsn_size_1):
        for j in range(wsn_size_2 - 1):
            C = f_prob[i, j, :]                                         # Node 1 of edge
            R = f_prob[i, j + 1, :]                                     # Node 2 of edge
            horizontal_psi[i, j, :, :] = ipf(T, R, C, num_of_values)    # edge potential
            pass

        pass

    # Vertical edge potentials
    vertical_psi = np.zeros((wsn_size_1 - 1, wsn_size_2, num_of_values, num_of_values))
    for i in range(wsn_size_1 - 1):
        for j in range(wsn_size_2):
            C = f_prob[i, j, :]                                     # Node 1 of edge
            R = f_prob[i + 1, j, :]                                 # Node 2 of edge
            vertical_psi[i, j, :, :] = ipf(T, R, C, num_of_values)  # edge potential
            pass
        pass

    return horizontal_psi, vertical_psi


# Functions for part 3:
def loopy_belief_propagation(test_sample, f_prob, horizontal_psi, vertical_psi, unique_values, num_of_values, terrain, ):
    """Runs LBP and estimates the values of the marked sensors. Then calculates the error of said estimations."""

    # Self potential * edge potential. Used for each direction of message in message passing.
    comb_left = np.einsum("ijk, ijkl -> ijl", f_prob[:, :-1, :], horizontal_psi)
    comb_right = np.einsum("ijk, ijkl -> ijl", f_prob[:, 1:, :], horizontal_psi)
    comb_up = np.einsum("ijk, ijkl -> ijl", f_prob[:-1, :, :], vertical_psi)
    comb_down = np.einsum("ijk, ijkl -> ijl", f_prob[1:, :, :], vertical_psi)

    # Initializations

    row = wsn_size_1
    col = wsn_size_2

    right = np.ones((row, col, num_of_values))
    left  = np.ones((row, col, num_of_values))
    up    = np.ones((row, col, num_of_values))
    down  = np.ones((row, col, num_of_values))

    right_new = np.ones((row, col, num_of_values))
    left_new  = np.ones((row, col, num_of_values))
    up_new    = np.ones((row, col, num_of_values))
    down_new  = np.ones((row, col, num_of_values))

    # Belief propagation
    for t in range(num_iters_lbp):
        # Message passing
        up_new[:-1, :, :]    = comb_up[:, :, :]    * up[1:, :, :]    * left[1:, :, :]  * right[1:, :, :]  + eps
        down_new[1:, :, :]   = comb_down[:, :, :]  * down[:-1, :, :] * left[:-1, :, :] * right[:-1, :, :] + eps
        left_new[:, 1:, :]   = comb_left[:, :, :]  * left[:, :-1, :] * up[:, :-1, :]   * down[:, :-1, :]  + eps
        right_new[:, :-1, :] = comb_right[:, :, :] * right[:, 1:, :] * up[:, 1:, :]    * down[:, 1:, :]   + eps

        # Normalization
        for i in range(wsn_size_1):
            for j in range(wsn_size_2):
                up[i, j, :]    = up_new[i, j, :]    / np.sum(up_new[i, j, :])
                down[i, j, :]  = down_new[i, j, :]  / np.sum(down_new[i, j, :])
                left[i, j, :]  = left_new[i, j, :]  / np.sum(left_new[i,
    ↪ j, :])
                right[i, j, :] = right_new[i, j, :] / np.sum(right_new[i,
    ↪  j, :])
                pass
            pass
        pass

    # Compute beliefs
    beliefs = f_prob * left * right * up * down

    # Create final map of moisture
    sens_off_idxs = np.where(terrain == False)
    sens_off_beliefs = unique_values[np.argmax(beliefs[sens_off_idxs], 1)
    ↪ ]
    estimated_sample = test_sample.copy()
    estimated_sample[sens_off_idxs] = sens_off_beliefs

    # Error measurements
    sample_difference = estimated_sample - test_sample
    num_sensors = row * col
    sum_mse = np.sum(np.power(sample_difference, 2))
    sum_mae = np.sum(np.abs(sample_difference))

    mse_lat = sum_mse / num_sensors
    mae_lat = sum_mae / num_sensors
    rmse_lat = np.sqrt(mse_lat)

    return mse_lat, mae_lat, rmse_lat, beliefs


def generate_graphs(beliefs_opt: np.ndarray, f_prob, horizontal_psi,
    ↪ vertical_psi, unique_values, num_of_values):
    """Runs LBP for different percentages of broken sensors and generates
    ↪  the graph values of the errors metrics."""

    # Percentage of sensors turned off
    sens_off = list(range(5, 80, 5))
    num_tests = len(sens_off)

    # Error metrics (lines)
    mae_list = np.zeros((num_tests, 1))
    mse_list = np.zeros((num_tests, 1))
    rmse_list = np.zeros((num_tests, 1))
    x_axis = np.asarray(sens_off)

    for i in range(num_tests):
        perc_off = sens_off[i]

        test_sample = beliefs_opt.copy()
        terrain = np.full((wsn_size_1, wsn_size_2), True)

        num_broken_sensors = wsn_size_1 * wsn_size_2 * perc_off // 100
```

```python
        broken_sensors = np.random.choice(wsn_size_1 * wsn_size_2, num_broken_sensors)

        # Terrain is a map of which sensors are functional or not.
        for broken_sensor in broken_sensors:
            x, y = divmod(broken_sensor, wsn_size_2)
            terrain[x, y] = False
            pass

        # Run LBP
        mse_list[i], mae_list[i], rmse_list[i], beliefs_est = loopy_belief_propagation(
            test_sample, f_prob, horizontal_psi, vertical_psi, unique_values, num_of_values, terrain)
        pass

    return x_axis, mse_list, mae_list, rmse_list



def main():
    global unique_values, num_of_values

    # For local use
    parse_console_arguments()

    t1_start = perf_counter()

    # Execution part 1: getting the useful information

    orig_moist_data = readers.read_np_array(shape=(31, 100, 100))
    samples, beliefs_opt = get_files_and_samples(orig_moist_data)
    moist = build_the_lattice_graph(samples)
    unique_values, num_of_values = get_classes(moist)

    # Execution part 2: Calculating the parts for the algorithms

    f_prob = get_f_prob(unique_values, num_of_values, moist)
    horizontal_psi, vertical_psi = get_edge_potentials(f_prob, num_of_values)

    # Execution part 3: Applying the algorithms

    x_axis, mse_list, mae_list, rmse_list = generate_graphs(
        beliefs_opt, f_prob, horizontal_psi, vertical_psi, unique_values, num_of_values)

    # Create the plot
    plt.plot(x_axis, mse_list, label="MSE")
    plt.plot(x_axis, rmse_list, label="RMSE")
    plt.plot(x_axis, mae_list, label="MAE")
    plt.xlabel("Percentage of broken sensors")
    plt.title("Loopy Belief Propagation on Lattice")
    plt.legend()

    # Save the plot
    writers.write_plt_fig(file_name="lbp_nasa", if_file_exists="overwrite
    ↪ ", fig_format="svg")

    print(f'Done, time elapsed: {round(perf_counter() - t1_start)}s')


if __name__ == '__main__':
    main()
```

Snippet A.2: Run LBP over on a dataset.

Appendix B

Screenshots from key parts of the website

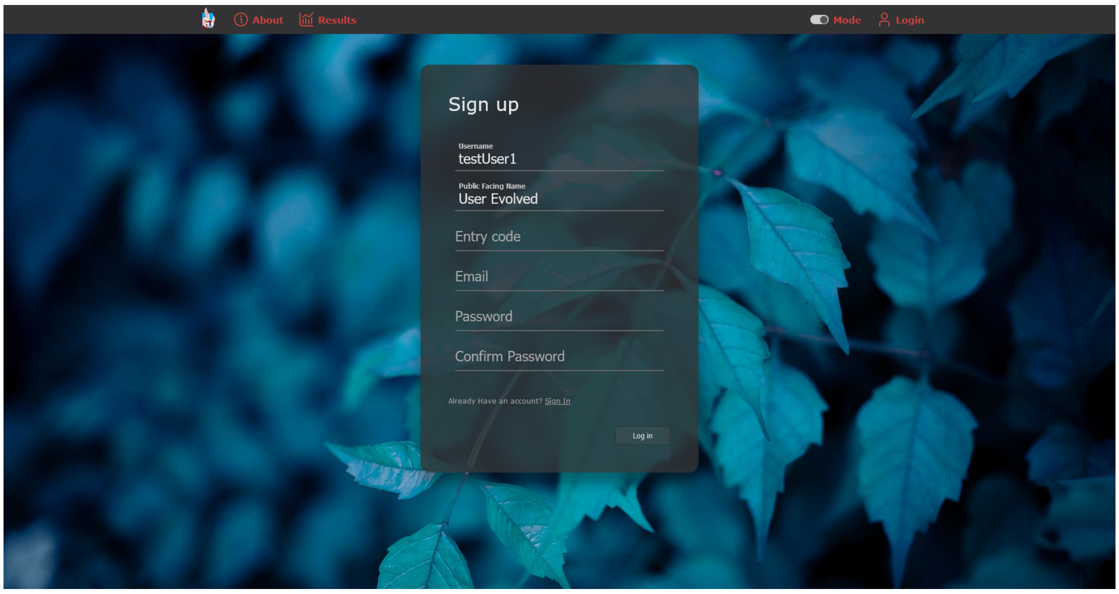


Welcome

This is the web app for the thesis **A platform to benchmark inference algorithms based on sensor network data**, by Thanassis Rentzepopoulos.

User logged in: User Evolved

Figure B.1: The home page.



Sign up

Username
testUser1

Public Facing Name
User Evolved

Entry code

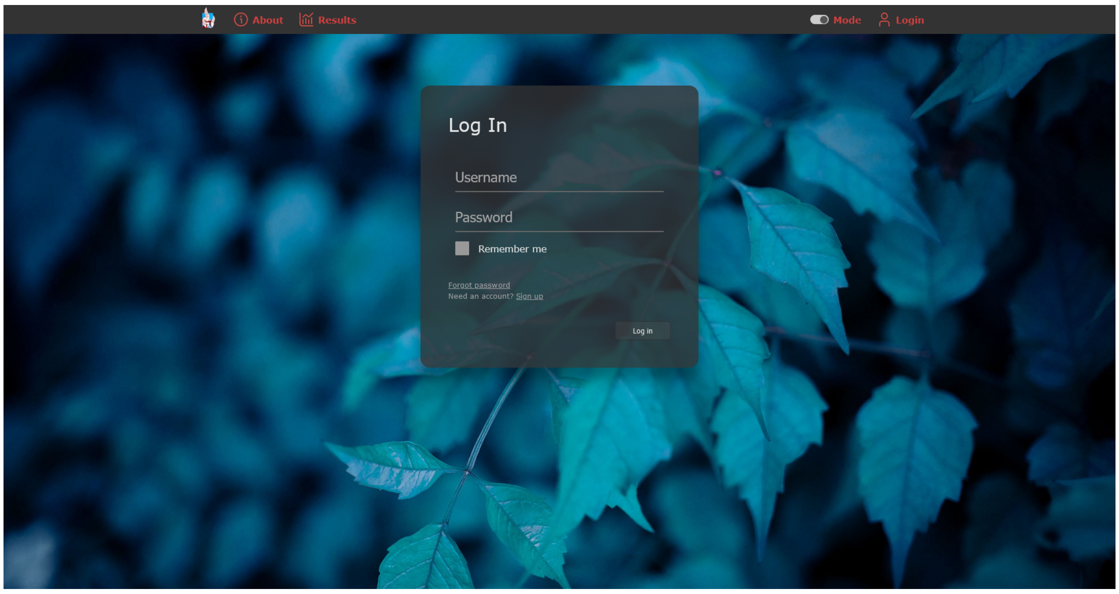
Email

Password

Confirm Password

Already Have an account? Sign In

Figure B.2: Signup. Note that to add a pseudo security measure, to register a user must know the ”Entry Code”.



Log In

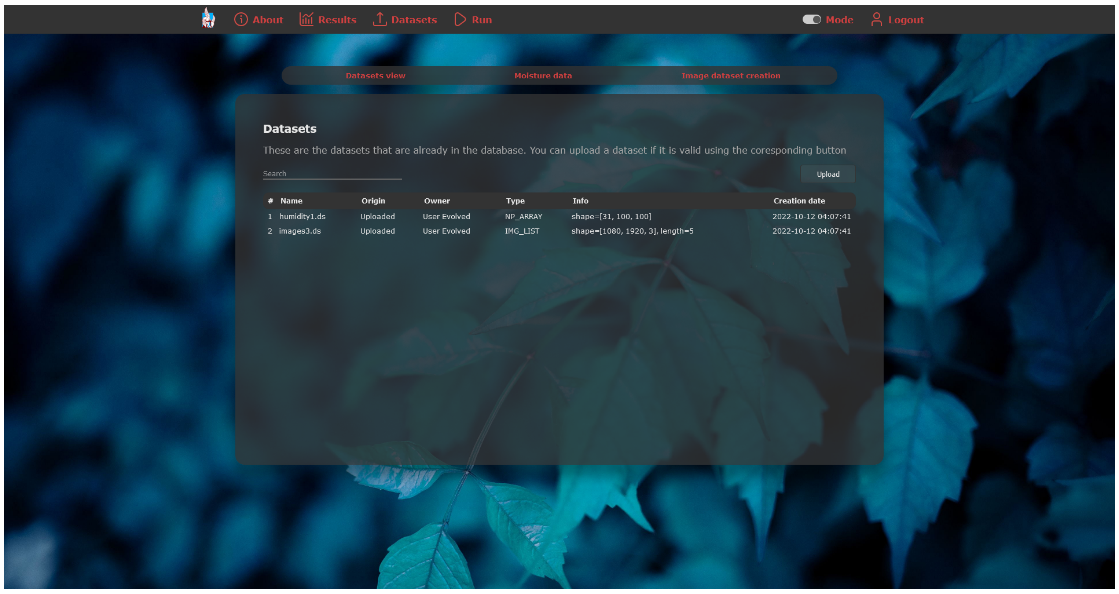
Username

Password

Remember me

Forgot password
Need an account? Sign up

Figure B.3: Login.

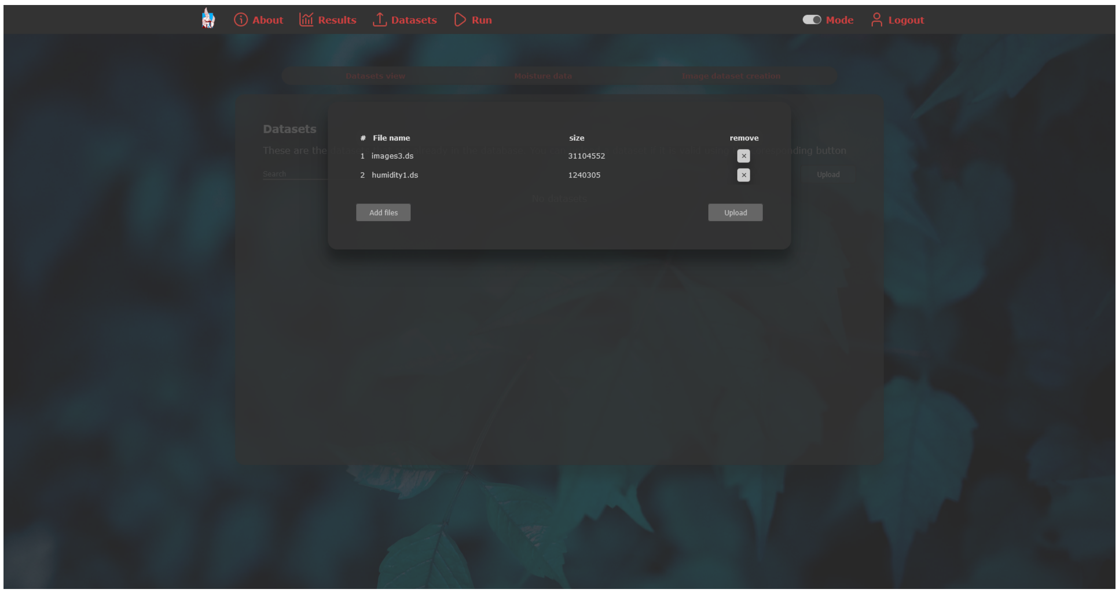


Datasets

These are the datasets that are already in the database. You can upload a dataset if it is valid using the coresponding button

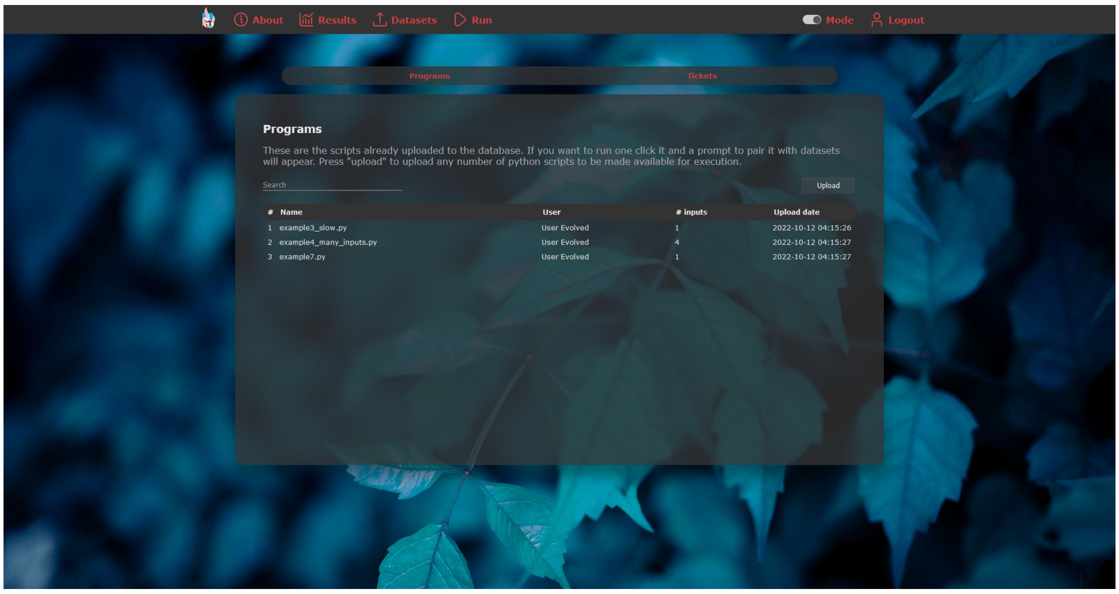
#	Name	Origin	Owner	Type	Info	Creation date
1	humidity1.ds	Uploaded	User Evolved	NP_ARRAY	shape=[31, 100, 100]	2022-10-12 04:07:41
2	images3.ds	Uploaded	User Evolved	IMG_LIST	shape=[1080, 1920, 3], length=5	2022-10-12 04:07:41

Figure B.4: Datasets page.



#	File name	size	remove
1	images3.ds	31104552	✕
2	humidity1.ds	1240305	✕

Figure B.5: Uploading datasets to the web app.

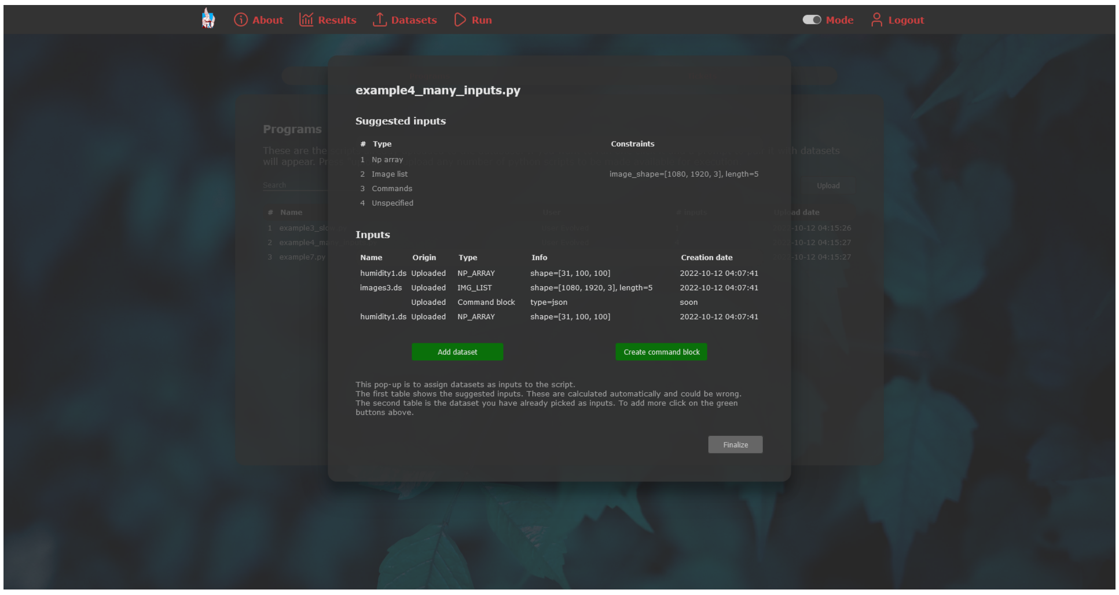


Programs

These are the scripts already uploaded to the database. If you want to run one click it and a prompt to pair it with datasets will appear. Press "upload" to upload any number of python scripts to be made available for execution.

#	Name	User	# inputs	Upload date
1	example3_slow.py	User Evolved	1	2022-10-12 04:15:26
2	example4_many_inputs.py	User Evolved	4	2022-10-12 04:15:27
3	example7.py	User Evolved	1	2022-10-12 04:15:27

Figure B.6: Programs' page. In here all scripts uploaded to the database are presented. Clicking on one opens a dialog to run it.



example4_many_inputs.py

Suggested inputs

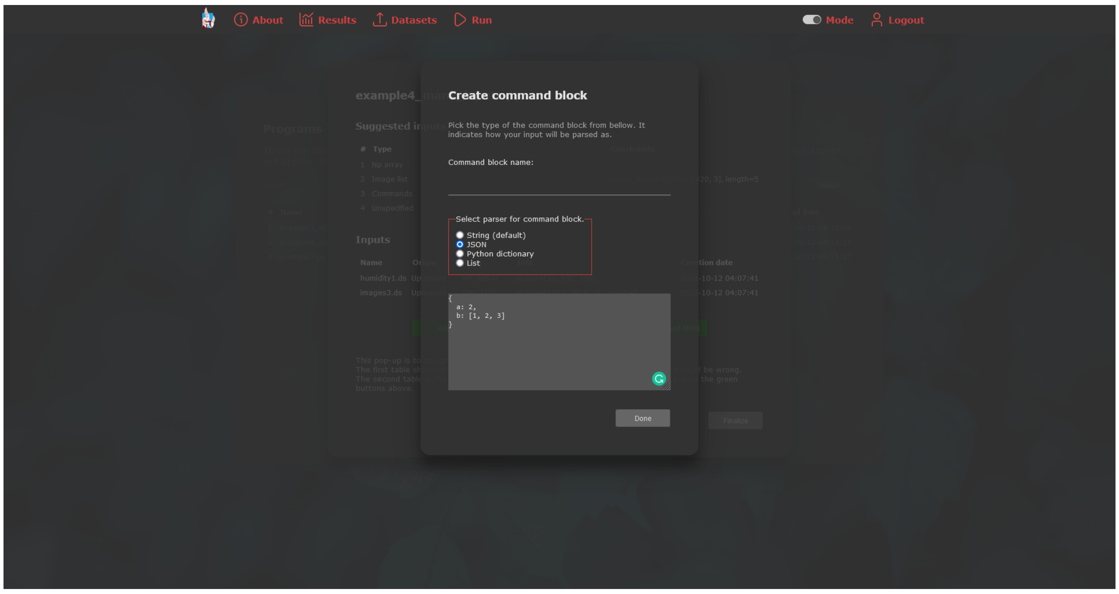
#	Type	Constraints
1	Np array	
2	Image list	image_shape=[1080, 1920, 3], length=5
3	Commands	
4	Unspecified	

Inputs

Name	Origin	Type	Info	Creation date
humidity1.ds	Uploaded	NP_ARRAY	shape=[31, 100, 100]	2022-10-12 04:07:41
images3.ds	Uploaded	IMG_LIST	shape=[1080, 1920, 3], length=5	2022-10-12 04:07:41
	Uploaded	Command block	type=json	soon
humidity1.ds	Uploaded	NP_ARRAY	shape=[31, 100, 100]	2022-10-12 04:07:41

This pop-up is to assign datasets as inputs to the script.
The first table shows the suggested inputs. These are calculated automatically and could be wrong.
The second table is the dataset you have already picked as inputs. To add more click on the green buttons above.

Figure B.7: Preparing a script to be run. The top table shows the suggested inputs as infered by the code analyzer. The bottom one holds the actual inputs selected.



Create command block

Pick the type of the command block from bellow. It indicates how your input will be parsed as.

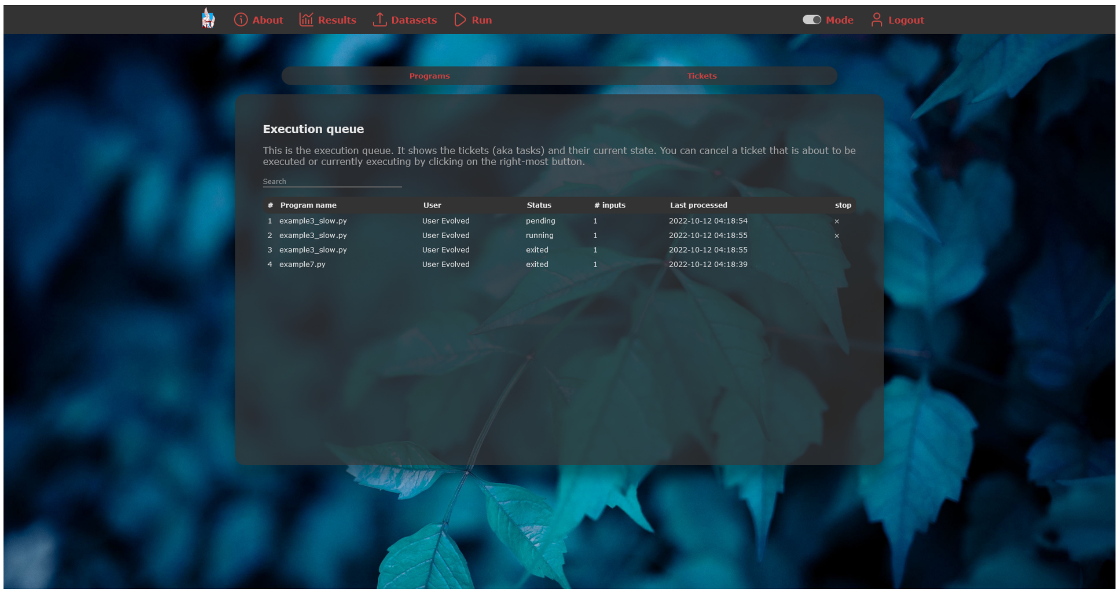
Command block name:

Select parser for command block.

( ) String (default)
[✓ JSON]
( ) Python dictionary
( ) List

```
{
  a: 2,
  b: [1, 2, 3]
}
```

Figure B.8: The UI portion of how a command block dataset gets created and to be passed to a script.

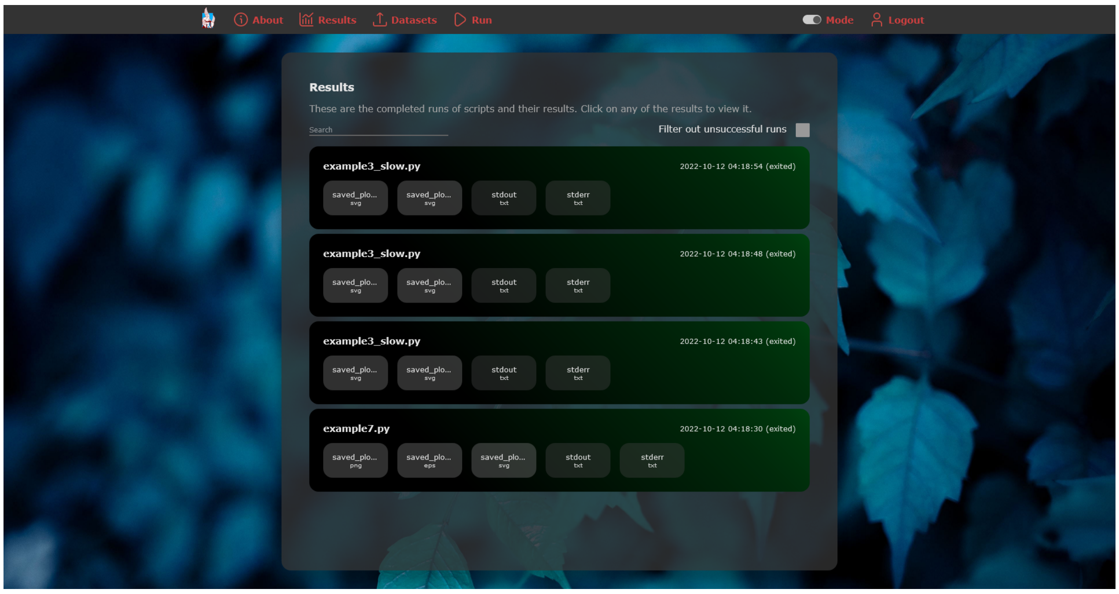


Execution queue

This is the execution queue. It shows the tickets (aka tasks) and their current state. You can cancel a ticket that is about to be executed or currently executing by clicking on the right-most button.

#	Program name	User	Status	# inputs	Last processed	stop
1	example3_slow.py	User Evolved	pending	1	2022-10-12 04:18:54	×
2	example3_slow.py	User Evolved	running	1	2022-10-12 04:18:55	×
3	example3_slow.py	User Evolved	exited	1	2022-10-12 04:18:55	
4	example7.py	User Evolved	exited	1	2022-10-12 04:18:39	

Figure B.9: The execution queue. Notice that programs that can be interrupted have a button for that next to them.



Results

These are the completed runs of scripts and their results. Click on any of the results to view it.

Search

Filter out unsuccessful runs

example3_slow.py 2022-10-12 04:18:54 (exited)
saved_plo... svg | saved_plo... svg | stdout txt | stderr txt

example3_slow.py 2022-10-12 04:18:48 (exited)
saved_plo... svg | saved_plo... svg | stdout txt | stderr txt

example3_slow.py 2022-10-12 04:18:43 (exited)
saved_plo... svg | saved_plo... svg | stdout txt | stderr txt

example7.py 2022-10-12 04:18:30 (exited)
saved_plo... png | saved_plo... eps | saved_plo... svg | stdout txt | stderr txt

Figure B.10: Results page. Successful runs are denoted with green and unsuccessful with red. Each button represents one of the outputs of the script.

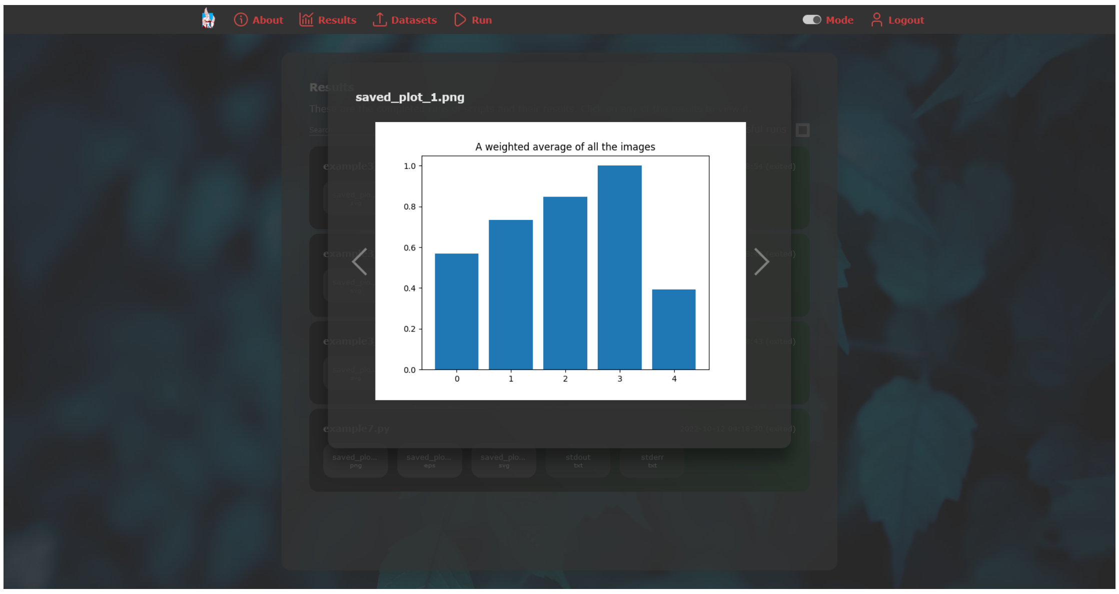


Figure B.11: Viewing a result.

Bibliography

[1] martinfowler.com. Microservices. https://martinfowler.com/articles/microservices.html. [Online; accessed 16-October-2022].

[2] Docker Inc. Use containers to build, share and run your applications. https://www.docker.com/resources/what-container/. [Online; accessed 16-October-2022].

[3] The Pallets Projects. Flask. https://palletsprojects.com/p/flask/. [Online; accessed 16-October-2022].

[4] Python 3.10.8 documentation. What is Python. https://docs.python.org/3/faq/general.html#what-is-python, . [Online; accessed 16-October-2022].

[5] MongoDB. Introduction to MongoDB. https://www.mongodb.com/docs/v6.0/introduction/. [Online; accessed 16-October-2022].

[6] RapidMiner. RapidMiner. https://rapidminer.com/. [Online; accessed 16-October-2022].

[7] Google. Colaboratory. https://research.google.com/colaboratory/faq.html. [Online; accessed 16-October-2022].

[8] Amazon. Get Started with Amazon SageMaker Notebook Instances. https://docs.aws.amazon.com/sagemaker/latest/dg/gs-console.html. [Online; accessed 16-October-2022].

[9] Microsoft. Learn more about all the notebooks experiences from Microsoft and GitHub. https://visualstudio.microsoft.com/vs/features/notebooks-at-microsoft/. [Online; accessed 16-October-2022].

[10] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, jun 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. URL https://doi.org/10.1145/564585.564601.

[11] Django Software Foundation. Django at a glance. https://docs.djangoproject.com/en/4.1/intro/overview/#django-at-a-glance. [Online; accessed 16-October-2022].

[12] tiangolo. FastAPI. https://github.com/tiangolo/fastapi. [Online; accessed 16-October-2022].

[13] Python documentation. Abstract Syntax Trees. https://docs.python.org/3/library/ast.html, . [Online; accessed 16-October-2022].

[14] Wei Zhao and Yao Liang. A systematic probabilistic approach to energy-efficient and robust data collections in wireless sensor networks. *IJSNet*, 7:162–175, 05 2010. doi: 10.1504/IJSNET.2010.033118.

[15] N. Friedman D. Koller. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.

List of Figures

List of Code Snippets